PODSTAWY INFORMATYKI

4/11/2019

WFAiS UJ, Informatyka Stosowana

I rok studiów, I stopień

Wykład 6 – część I

Modele danych: zbiory

- Podstawowe definicje
- Operacje na zbiorach
- Prawa algebraiczne
- Struktury danych
  - Lista jednokierunkowa
  - Wektor własny
  - Tablica mieszająca

Zbiór

- Zbiór jest najbardziej podstawowym modelem danych w matematyce. Wszystkie pojęcia matematyczne, od drzew po liczby rzeczywiste można wyrazić za pomocą specjalnego rodzaju zbioru.
- Jest wiec naturalne że jest on również podstawowym modelem danych w informatyce.
- Dotychczas wykorzystaliśmy to pojęcie mówiąc o słowniku, który także jest rodzajem zbioru na którym możemy wykonywać tylko określone operacje: wstawiania, usuwania i wyszukiwania.

Podstawowe definicje

- W matematyce pojęcie zbioru nie jest zdefiniowane wprost.
- Zamiast tego, podobnie jak punkt czy prosta w geometrii, zbiór jest zdefiniowany za pomocą swoich własności.
- W szczególności istnieje pojęcie przynależności, które jest sensowne tylko i wyłącznie dla zbiorów. Jeśli S jest zbiorem oraz x jest czymkolwiek, zawsze możemy odpowiedzieć na pytanie

  „Czy x należy do zbioru S?"
- Zbiór S składa się więc z wszystkich takich elementów x, dla których x należy do zbioru S.

Podstawowe definicje

- **Notacja:**
  - Wyrażenie $x \in S$ oznacza, że element x należy do zbioru S.
  - Jeśli elementy $x_1, x_2, \ldots, x_n$ należą do zbioru S i żadne inne, to możemy zapisać:
    $S = \{x_1, x_2, \ldots, x_n\}$
  - Każdy x musi być inny, nie możemy umieścić w zbiorze żadnego elementu dwa lub więcej razy. Kolejność ułożenia elementów w zbiorze jest jednak całkowicie dowolna.
  - Zbiór pusty, oznaczamy symbolem $\emptyset$, jest zbiorem do którego nie należą żadne elementy.
    - Oznacza to że $x \in \emptyset$ jest zawsze fałszywe.

Podstawowe definicje

- **Definicja za pomocą abstrakcji:**
  - **Wyliczenie elementów należących do zbioru nie jest jedynym sposobem jego definiowania. Bardzo wygodne jest wyjście od definicji że istnieje zbiór S oraz że jego elementy spełniają własność P, tzn. {x : x ∈ S oraz P(x) } czyli „zbiór takich elementów x należących do zbioru S, które spełniają własność P.**
- **Równość zbiorów:**
  - **Dwa zbiory są równe (czyli są tym samym zbiorem), jeśli zawierają te same elementy.**
- **Zbiory nieskończone:**
  - **Zwykle wygodne jest przyjęcie założenia że zbiory są skończone. Czyli że istnieje pewna skończona liczba N taka, że nasz zbiór zawiera dokładnie N elementów. Istnieją jednak również zbiory nieskończone np. liczb naturalnych, całkowitych, rzeczywistych, itd.**

Operacje na zbiorach

Operacje często wykonywane na zbiorach:

- **Suma:** dwóch zbiorów S i T, zapisywana S ∪ T, czyli zbiór zawierający elementy należące do zbioru S lub do zbioru T.
- **Przecięcie (iloczyn):** dwóch zbiorów S i T, zapisywana S ∩ T, czyli zbiór zawierający należące elementy do zbioru S i do zbioru T.
- **Różnica:** dwóch zbiorów S i T, zapisywana S \ T, czyli zbiór zawierający tylko te elementy należące do zbioru S, które nie należą do zbioru T.

Operacje na zbiorach

- **Jeżeli S i T są zdarzeniami w przestrzeni probabilistycznej,**
  - **suma, przecięcie i różnica mają naturalne znaczenie,**
  - **S ⋃ T jest zdarzeniem polegającym na zajściu zdarzenia S lub T,**
  - **S ⋂ T jest zdarzeniem polegającym na zajściu zdarzenia S i T,**
  - **S \ T jest zdarzeniem polegającym na zajściu zdarzenia S ale nie T,**
  - **Jeśli S jest zbiorem obejmującym całą przestrzeń probabilistyczna, S \ T jest dopełnieniem zbioru T.**

Prawa algebraiczne

- **Prawo przemienności i łączności dla sumy zbiorów określają, że możemy obliczyć sumę wielu zbiorów, wybierając je w dowolnej kolejności.**
- **Wynikiem zawsze będzie taki sam zbiór elementów, czyli takich które należą do jednego lub więcej zbiorów będących operandami sumy.**

Prawo przemienności dla sumy:

$$(S \cup T) = (T \cup S)$$

Prawo łączności dla sumy:

$$(S \cup (T \cup R)) = ((S \cup T) \cup R)$$

Prawo przemienności dla przecięcia:

$$(S \cap T) = (T \cap S)$$

Prawo łączności dla przecięcia:

$$(S \cap (T \cap R)) = ((S \cap T) \cap R)$$

Prawa algebraiczne

- Przecięcie dowolnej liczby zbiorów nie zależy od kolejności ich grupowania

Prawo rozdzielności przecięcia względem sumy:

$$(S \cap (T \cup R)) = ((S \cap T) \cup (S \cap R))$$

Prawo rozdzielności sumy względem przecięcia:

$$(S \cup (T \cap R)) = ((S \cup T) \cap (S \cup R))$$

Prawa algebraiczne

Prawo łączności dla sumy i różnicy:

$(S \setminus (T \cup R)) = ((S \setminus T) \cup (S \setminus R))$

Prawo rozdzielności różnicy względem sumy:

$((S \cup T) \setminus R) = ((S \setminus R) \cup (T \setminus R))$

- Zbiór pusty jest elementem neutralnym sumy: $(S \cup \varnothing) \equiv S$
- Idempotencja sumy: $(S \cup S) = S$
- Idempotencja przecięcia: $(S \cap S) = S$
- $(S \setminus S) \equiv \varnothing$
- $(\varnothing \setminus S) \equiv \varnothing$
- $(\varnothing \cap S) \equiv \varnothing$

Prawa algebraiczne

- **Relacja podzbioru:**
  - Istnieje relacja zawierania się jednego zbioru w drugim zbiorze, co oznacza że wszystkie elementy pierwszego są również elementami drugiego.
- **Zbiór potęgowy:**
  - P(S) zbioru S to zbiór wszystkich podzbiorów zbioru S.
  - Jeśli S = {1,2,3} to:
  - P(S) = {∅, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3} }.

Obiekty niepodzielne

- **To nie jest pojęcie z teorii zbiorów ale bardzo wygodne dla dyskusji o strukturach danych i algorytmach opartych na zbiorach.**
- **Zakładamy istnienie pewnych obiektów niepodzielnych, które nie są zbiorami. Obiektem niepodzielnym może być element zbioru, jednak nic nie może należeć do samego obiektu niepodzielnego. Podobnie jak zbiór pusty, obiekt niepodzielny nie może zawierać żadnych elementów. Zbiór pusty jest jednak zbiorem, obiekt niepodzielny nim nie jest.**
- **Kiedy mówimy o strukturach danych, często wygodne jest wykorzystanie skomplikowanych typów danych jako typów obiektów niepodzielnych. Obiektami niepodzielnymi mogą wiec być struktury lub tablice, które przecież wcale nie mają „niepodzielnego" charakteru.**

Implementacja zbioru na strukturze listy

- **Suma, przecięcie i różnica:**
  - Podstawowe operacje na zbiorach, np. suma, mogą wykorzystywać jako przetwarzaną strukturę danych listę jednokierunkową, chociaż właściwa technika przetwarzania zbiorów powinna się nieco różnić od stosowanej przez nas dla list.
  - W szczególności posortowanie elementów wykorzystywanych list znacząco skraca czas wykonywania operacji sumy, przecięcia i różnicy zbiorów.

Zbiory a listy

- **Istotne różnice miedzy pojęciami:**
  - **zbiór $S = \{x_1, x_2, \ldots, x_n\}$**
  - **lista $L = (x_1, x_2, \ldots, x_n)$**

  - **Kolejność elementów w zbiorze jest nieistotna (a dla listy jest istotna).**

  - **Elementy należące do zbioru nie mogą się powtarzać (a dla listy mogą).**

Zbiory a listy

Zbiory jako nieposortowane listy:

- Wyznaczenie sumy, przecięcia czy różnicy zbiorów o rozmiarach m i n wymaga czasu O(m n).
  - Aby stworzyć listę U reprezentująca np. sumę pary S i T, musimy rozpocząć od skopiowania listy reprezentującej zbiór S do początkowo pustej listy U.
  - Następnie każdy element listy ze zbioru T musimy sprawdzić aby przekonać się, czy nie znajduje się on na liście U.
  - Jeśli nie to dodajemy ten element do listy U.

Zbiory a listy

Zbiory jako posortowane listy

- **Operacje wykonujemy znacznie szybciej jeżeli elementy są posortowane. Za każdym razem porównujemy ze sobą tylko dwa elementy (po jednym z każdej listy).**
  - **Wyznaczenie sumy, przecięcia czy różnicy zbiorów o rozmiarach m i n wymaga czasu $O(m+n)$.**
  - **Jeżeli listy nie były pierwotnie posortowane to sortowanie list zajmuje $O(m \log m + n \log n)$.**
  - **Operacja ta może nie być szybsza niż $O(m\, n)$ jeśli ilość elementów list jest bardzo różna.**
  - **Jeżeli liczby m i n są porównywalne to $O(m \log m + n \log n) < O(m\, n)$**

Implementacja zbiorów oparta na wektorze własnym

- Definiujemy uniwersalny zbiór U w którym zawierają się wszystkie zbiory na których będziemy przeprowadzać operacje. Np. talia kart , czyli zbiór 52 kart, jest uniwersalnym zbiorem dla różnych możliwych zbiorów kart.
- Porządkujemy elementy zbioru U w taki sposób, by każdy element tego zbioru można było związać z unikatową „pozycją ", będącą liczbą całkowitą od 0 do N-1 (gdzie N jest liczba elementów w zbiorze uniwersalnym). Liczba elementów w zbiorze S jest m.
- Wówczas, zbiór S zawierający się w zbiorze U, możemy reprezentować za pomocą wektora własnego złożonego z zer i jedynek – dla każdego elementu x należącego do zbioru U, jeśli x należy także do zbioru S, odpowiadająca temu elementowi pozycja zawiera wartość 1; jeśli x nie należy do S, na odpowiedniej pozycji mamy wartość 0.

Implementacja zbiorów oparta na wektorze własnym

- **Czas potrzebny na wykonanie operacji sumy, przecięcia i różnicy jest O(N).**
- **Jeśli przetwarzane zbiory są dużą częścią zbioru uniwersalnego to jest to dużo lepsze niż O(m log m) (posortowanie listy) lub $O(m^2)$ (nieposortowane listy).**
- **Jeśli m << N to jest to oczywiście nieefektywne.**
- **Ta implementacja również niepraktyczna jeżeli wymaga zbyt dużego U.**

Przykład z kartami

- **Przykład:**
  - Niech U będzie talią kart.
  - Umawiamy się że porządkujemy karty w talii w następujący sposób:
    - Kolorami: trefl, karo, kier, pik.
    - W każdym kolorze wg schematu: as, 2, 3, …, walet, dama, król.
  - Przykładowo pozycja:
    - as trefl to 0,
    - król trefl to 12,
    - as karo to 13,
    - walet pik to 49.

Przykład z kartami

- Zbiór wszystkich kart koloru trefl
  - 1111111111111000000000000000000000000000000000000000
- Zbiór wszystkich figur
  - 0000000000111000000000011100000000001110000000000111
- Poker w kolorze kier (as, walet, dama, król)
  - 0000000000000000000000000010000000001110000000000000
- Każdy element zbioru kart jest związany z unikatową pozycją.

Przykład z jabłkami

	Odmiana	Kolor	Dojrzewa
0	Delicious	czerwony	późno
1	Granny Smith	zielony	wcześnie
2	Jonathan	czerwony	wcześnie
3	McIntosh	czerwony	wcześnie
4	Gravenstein	czerwony	późno
5	Pippin	zielony	późno

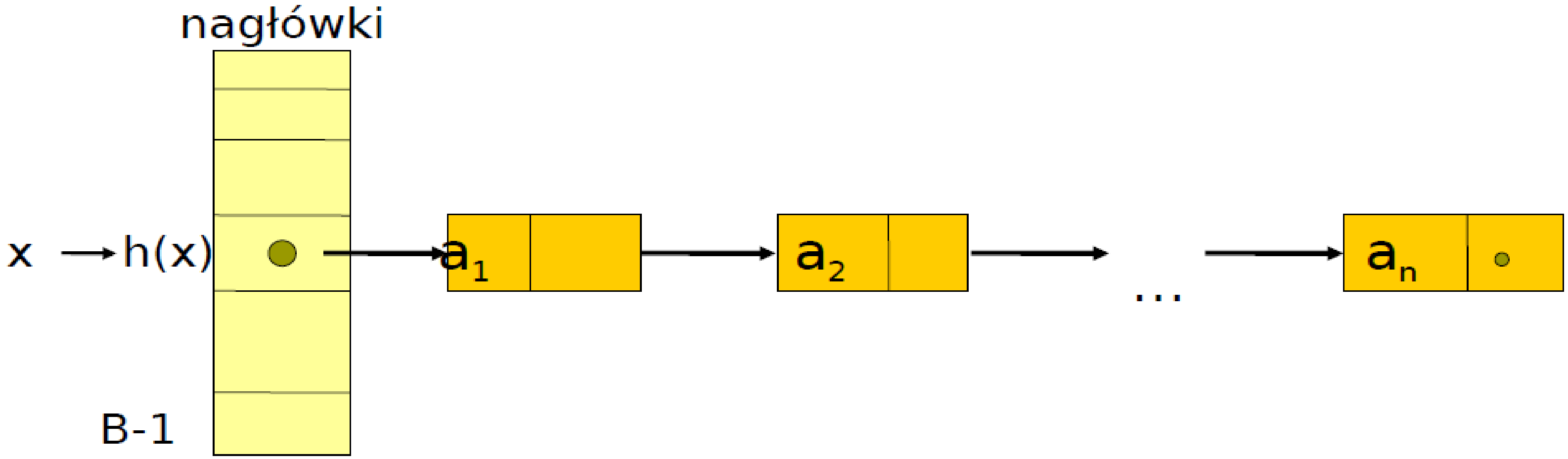
Czerwone = 101110
Wcześnie = 011100
Czerwone $\bigcup$ Wcześnie = 111110
Czerwone $\bigcap$ Wcześnie = 001100

Czas potrzebny do wyznaczenia sumy, przecięcia i różnicy jest proporcjonalny do długości N wektora własnego.

Struktura danych oparta na tablicy mieszającej

- **Reprezentacja słownika oparta o wektor własny, jeśli tylko możliwa, umożliwiłaby bezpośredni dostęp do miejsca w którym element jest reprezentowany.**
- **Nie możemy jednak wykorzystywać zbyt dużych zbiorów uniwersalnych ze względu na pamięć i czas inicjalizacji.**
- **Np. słownik dla słów złożonych z co najwyżej 10 liter.**
  Ile możliwych kombinacji: $26^{10}+26^{9}+ \ldots + 26 = 10^{14}$ możliwych słów.
  Faktyczny słownik: to tylko około 10^{6}.
  Co robimy?
- **Grupujemy, każda grupa to jedna komórka z „nagłówkiem” + lista jednokierunkowa z elementami należącymi do grupy.**
- **Taką strukturę nazywamy tablicą mieszająca (ang. hash table)**

Struktura danych tablicy mieszającej



- Istnieje funkcja mieszającą (ang. hash function), która jako argument pobiera element x i zwraca liczbę całkowitą z przedziału 0 do B-1, gdzie B jest liczbą komórek w tablicy mieszającej.
- Wartością zwracaną przez h(x) jest index komórki w której jest wskaźnik do listy, w której umieszczamy element x.
- Ważne aby funkcja h(x) „mieszała", tzn. aby listy zawierały tę samą przybliżoną liczbę elementów.

Struktura danych tablicy mieszającej

- **Każda komórka składa się z wskaźnika do listy jednokierunkowej, w której przechowujemy wszystkie elementy zbioru wysłanego do tej komórki przez funkcje mieszającą.**
- **Aby odnaleźć element x obliczamy wartość h(x), która wskazuje na numer komórki.**
- **Jeśli tablica mieszająca zawiera element x, to możemy go znaleźć przeszukując listę która znajduje się w tej komórce.**
- **Tablica mieszająca pozwala na wykorzystanie reprezentacji zbiorów opartej na liście (wolne przeszukiwanie), ale dzięki podzieleniu zbioru na B komórek, czas przeszukiwania jest ~ 1/B potrzebnego do przeszukiwania całego zbioru.**
- **W szczególności może być nawet O(1), czyli taki jak w reprezentacji zbioru opartej na wektorze własnym.**

Implementacja słownika oparta na tablicy mieszającej

- **Aby wstawić, usunąć lub wyszukać element x w słowniku zaimplementowanym przy użyciu tablicy mieszającej, musimy zrealizować proces złożony z trzech kroków:**
  - Wyznaczyć właściwą komórkę przy użyciu funkcji h(x)
  - Pobrać wskaźnik listy elementów znajdującej się w komórce wskazanej przez h(x)
  - Wykonać na tej liście operacje tak jakby reprezentowała cały zbiór

Podsumowanie

- **Pojęcie zbioru ma zasadnicze znaczenie w informatyce.**
- **Najczęściej wykonywanymi operacjami na zbiorach są: suma, przecięcie oraz różnica.**
- **Do modyfikowania i upraszczania wyrażeń złożonych ze zbiorów i zdefiniowanych na nich operacji możemy wykorzystywać prawa algebraiczne.**

Podsumowanie

- Listy jednokierunkowe, wektory własne oraz tablice mieszające to trzy najprostsze sposoby reprezentowania zbiorów w języku programowania.
- Listy jednokierunkowe oferują największą elastyczność w przypadku większości operacji na zbiorach, nie zawsze są jednak rozwiązaniem najbardziej efektywnym.
- Wektory własne są najszybszym rozwiązaniem dla pewnych operacji, mogą jednak być wykorzystywane tylko w sytuacjach, gdy zbiór uniwersalny jest mały.
- Często złotym środkiem są tablice mieszające, które zapewniają zarówno oszczędne wykorzystanie pamięci jak i satysfakcjonujący czas wykonania operacji.

Wykład 6 – część II

Modele danych: drzewa

- Podstawowa terminologia
- Rekurencyjna definicja
- Drzewa zaetykietowane
  - Drzewa wyrażeń
- Struktura danych dla drzew
  - Reprezentacje drzewa
  - Rekurencja w drzewach
- Drzewa binarne
- Drzewa przeszukiwania binarnego
- Drzewa binarne częściowo uporządkowane
- Zrównoważone drzewa częściowo uporządkowane
- Kopce i sortowanie przez kopcowanie

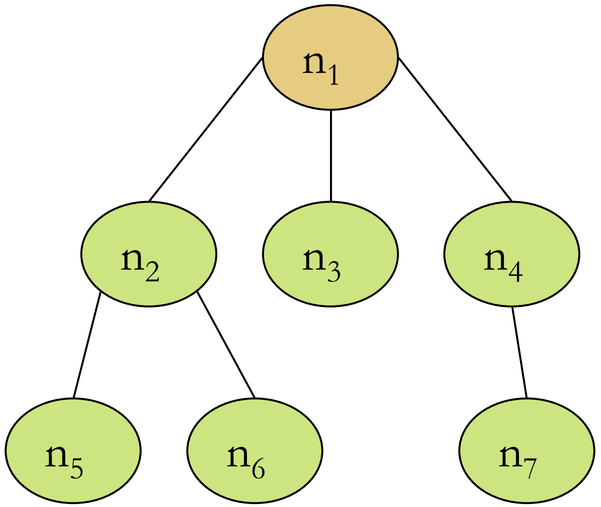
Model danych oparty na drzewach

- **Istnieje wiele sytuacji w których przetwarzane informacje mają strukturę hierarchiczną lub zagnieżdżoną, jak drzewo genealogiczne lub diagram struktury organizacyjnej.**

- **Abstrakcje modelujące strukturę hierarchiczną nazywamy drzewem – jest to jeden z najbardziej podstawowych modeli danych w informatyce.**

Podstawowa terminologia

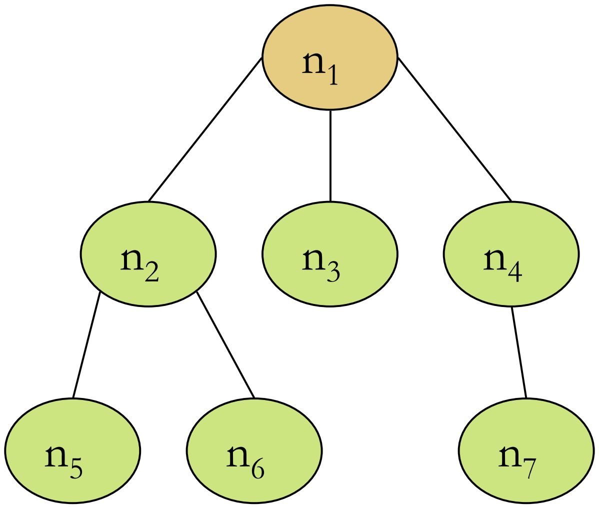
- **Drzewa są zbiorami punktów, zwanych węzłami lub wierzchołkami, oraz połączeń, zwanych krawędziami.**
- **Krawędź łączy dwa różne węzły.**



n_1 = rodzic n_2, n_3, n_4
n_2 = rodzic n_5, n_6
n_6 = dziecko n_2

Podstawowa terminologia

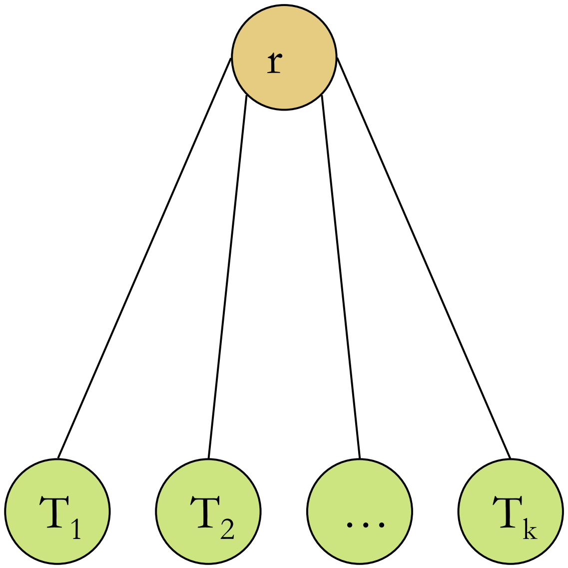
- **Aby struktura zbudowana z węzłów połączonych krawędziami była drzewem musi spełniać pewne warunki:**
  - **W każdym drzewie wyróżniamy jeden węzeł zwany korzeniem n_1 (ang. root)**
  - **Każdy węzeł c nie będący korzeniem jest połączony krawędzią z innym węzłem zwanym rodzicem p (ang. parent) węzła c. Węzeł c nazywamy także dzieckiem (ang. child) węzła p.**
  - **Każdy węzeł c nie będący korzeniem ma dokładnie jednego rodzica.**
  - **Każdy węzeł ma dowolną liczbę dzieci.**
  - **Drzewo jest spójne (ang. connected) w tym sensie że jeżeli rozpoczniemy analizę od dowolnego węzła c nie będącego korzeniem i przejdziemy do rodzica tego węzła, następnie do rodzica tego rodzica, itd., osiągniemy w końcu korzeń.**



n_1 = rodzic n_2, n_3, n_4
n_2 = rodzic n_5, n_6
n_6 = dziecko n_2

Rekurencyjna definicja drzew

- **Podstawa:** Pojedynczy węzeł n jest drzewem. Mówimy że n jest korzeniem drzewa złożonego z jednego węzła.
- **Indukcja:** Niech r będzie nowym węzłem oraz niech T_1, T_2, …, T_k będą drzewami zawierającymi odpowiednio korzenie c_1, c_2, …, c_k. Załóżmy że żaden węzeł nie występuje więcej niż raz w drzewach T_1, T_2, …, T_k, oraz że r, będący „nowym" węzłem, nie występuje w żadnym z tych drzew. Nowe drzewo T tworzymy z węzła r i drzew T_1, T_2, …, T_k w następujący sposób:
  - węzeł r staje się korzeniem drzewa T;
  - dodajemy k krawędzi, po jednej łącząc r z każdym z węzłów c_1, c_2, …, c_k, otrzymując w ten sposób strukturę w której każdy z tych węzłów jest dzieckiem korzenia r. Inny sposób interpretacji tego kroku to uczynienie z węzła r rodzica każdego z korzeni drzew T_1, T_2, …, T_k.

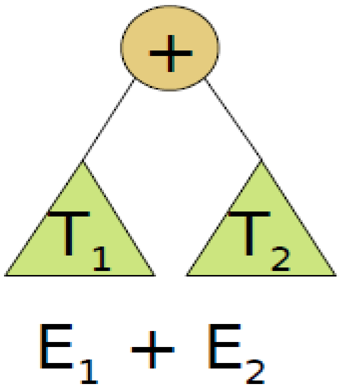


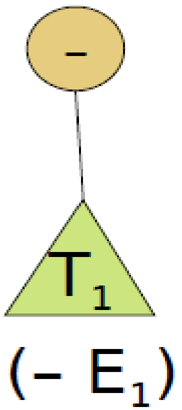
Podstawowa terminologia

- Relacje rodzic-dziecko można w naturalny sposób rozszerzyć do relacji **przodków i potomków**.
- **Ścieżką** nazywamy ciąg węzłów, takich że poprzedni jest rodzicem następnego. Węzły na ścieżce to potomkowie (przodkowie). Jeżeli ciąg węzłów (n_1, n_2, …, n_k) jest ścieżka, to **długość ścieżki** wynosi **k-1** (długość ścieżki dla pojedynczego węzła wynosi 0). Jeżeli ścieżka ma długość **>=1**, to węzeł m1 nazywamy właściwym przodkiem węzła m_k, a węzeł m_k właściwym potomkiem węzła m_1.
- W dowolnym drzewie **T**, dowolny węzeł **n** wraz z jego potomkami nazywamy **poddrzewem**.
- **Liściem** (ang. leaf) nazywamy węzeł drzewa który nie ma potomków.
- **Węzeł wewnętrzny** to taki węzeł który ma jednego lub większą liczbę potomków.
- **Wysokość drzewa** to długość najdłuższej ścieżki od korzenia do liścia.
- **Głębokość węzła** to długość drogi od korzenia do tego węzła.

Drzewa zaetykietowane i drzewa wyrażeń

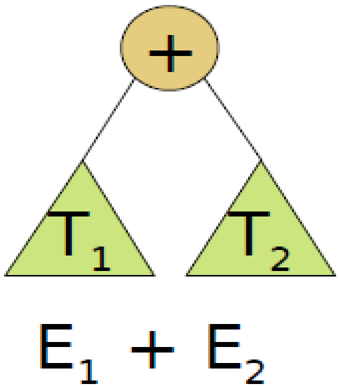
- **Drzewo zaetykietowane** to takie w którym z każdym węzłem drzewa związana jest jakaś etykieta lub wartość. Możemy reprezentować wyrażenia matematyczne za pomocą drzew zaetykietowanych.

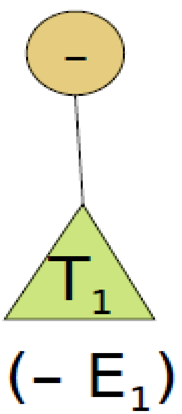




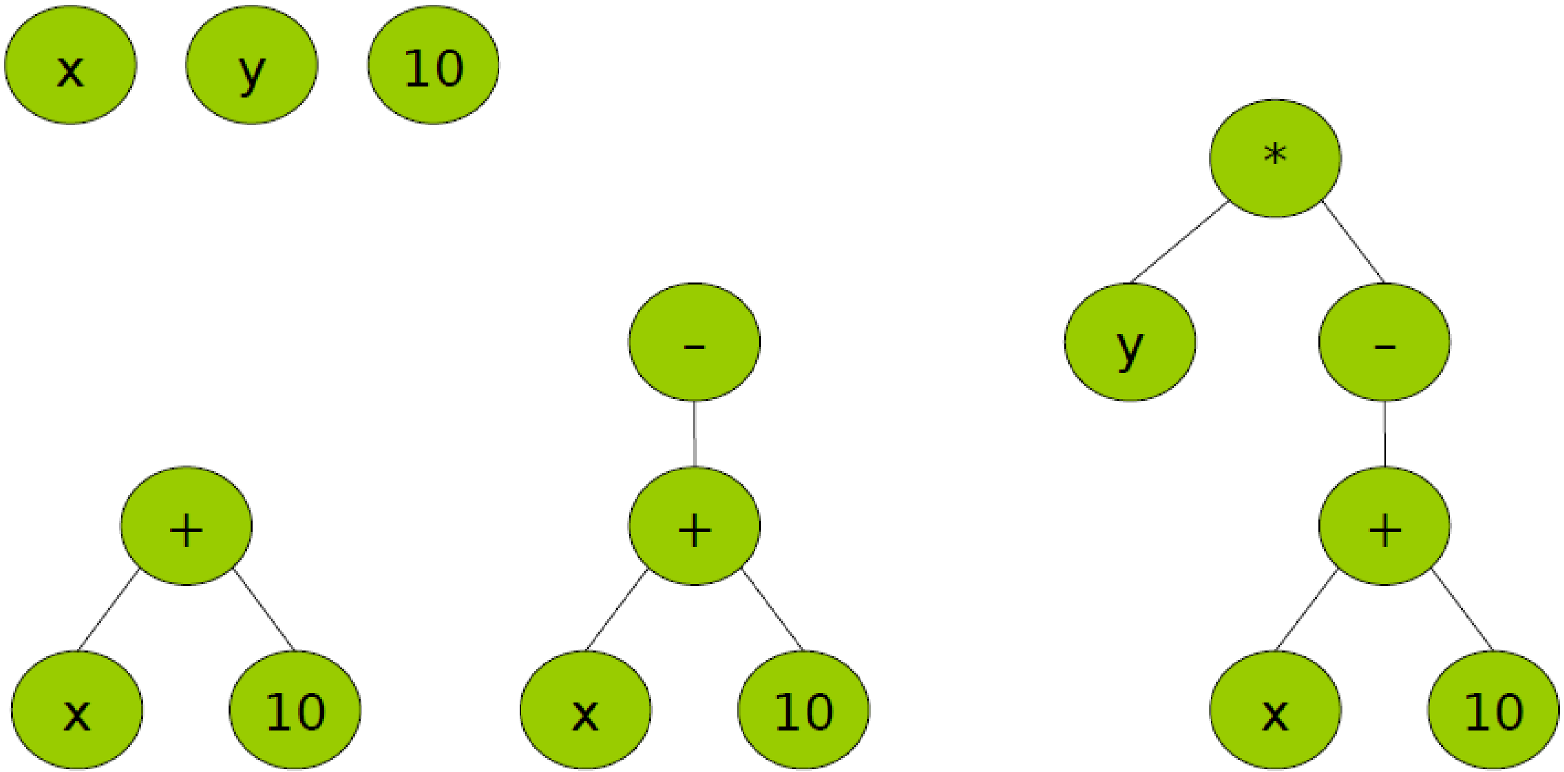
Drzewa zaetykietowane i drzewa wyrażeń

- **Definicja drzewa zaetykietowanego dla wyrażeń arytmetycznych zawierających operandy dwuargumentowe +,-,•,/ oraz operator jednoargumentowy -.**
  - **Podstawa:** **Pojedynczy operand niepodzielny jest wyrażeniem. Reprezentujące go drzewo składa się z pojedynczego węzła, którego etykietą jest ten operand.**
  - **Indukcja:** **Jeśli E1 oraz E2 są wyrażeniami reprezentowanymi odpowiednio przez drzewa T1, T2, wyrażenie (E1 + E2) reprezentowane jest przez drzewo którego korzeniem jest węzeł o etykiecie +. Korzeń ten ma dwoje dzieci, którego korzeniami są odpowiednio korzenie drzew T1, T2.**





Konstrukcja drzew wyrażeń



(x + 10)   (- (x + 10))   (y * - (x + 10))

Struktura danych dla drzew

- Do reprezentowania drzew możemy używać wiele różnych struktur danych. Wybór odpowiedniej struktury zależy od konkretnych operacji które planujemy wykonać na budowanych drzewach.
- Przykład:
  - Jeżeli jedynym planowanym działaniem jest lokalizowanie rodziców danych węzłów, zupełnie wystarczającą będzie struktura składająca się z etykiety węzła i wskaźnika do struktury reprezentującej jego rodzica.
- W ogólności, węzły drzewa możemy reprezentować za pomocą struktur, których pola łączą węzły w drzewa w sposób podobny do łączenia węzła za pomocą wskaźnika do struktury korzenia.

Struktura danych dla drzew

- Kiedy mówimy o reprezentowaniu drzew, w pierwszej kolejności mamy na myśli sposób reprezentowania węzłów.
- Różnica miedzy reprezentacjami dotyczy miejsca w pamięci komputera gdzie przechowywana jest struktura zawierająca węzły.
  - W języku C możemy stworzyć przestrzeń dla struktur reprezentujących wierzchołki za pomocą funkcji malloc ze standartowej biblioteki stdhlib.h, co powoduje, że do umieszczonych w pamięci węzłów mamy dostęp tylko za pomocą wskaźników.
  - Rozwiązaniem alternatywnym jest stworzenie tablicy struktur i wykorzystanie jej elementów do reprezentowania węzłów. Możemy uzyskać dostęp do węzłów nie wykorzystując ścieżek w drzewie. Wadą jest z góry określony rozmiar tablicy (musi istnieć ograniczenie maksymalnego rozmiaru drzewa).

Reprezentacja drzewa przy wykorzystaniu tablicy wskaźników

- **Jednym z najprostszych sposobów reprezentowania drzewa jest wykorzystanie dla każdego węzła struktury składającej się z pola lub pól reprezentujących etykietę oraz tablicy wskaźników do dzieci tego węzła.**
- **Info reprezentuje etykietę węzła.**
- **Stała bf jest rozmiarem tablicy wskaźników. Reprezentuje maksymalną liczbę dzieci dowolnego węzła, czyli czynnik rozgałęzienia (ang. branching factor).**
- **i-ty element tablicy reprezentującej węzeł zawiera wskaźnik do i-tego dziecka tego węzła.**
- **Brakujące połączenia możemy reprezentować za pomocą wskaźnika pustego NULL.**

```
typedef struct NODE *pNODE
struct NODE{
    int info;
    pNODE children[BF];
};
```

info			
p_0	p_1	…	P_{bf-1}

Reprezentacja drzewa przy wykorzystaniu listy jednokierunkowej

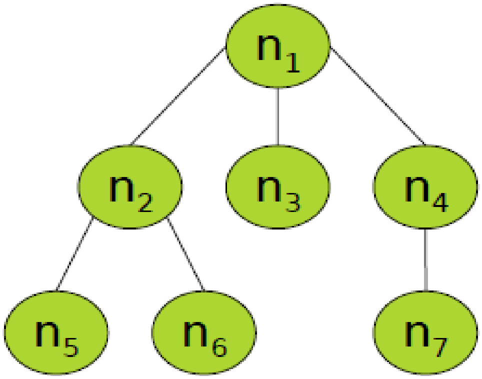
- Wykorzystujemy listę jednokierunkową reprezentującą dzieci węzła. Przestrzeń zajmowana przez listę jest dla węzła proporcjonalna do liczby jego dzieci.
- Znaczącą wadą tego rozwiązania jest efektywność czasowa – uzyskanie dostępu do i-tego dziecka wymaga czasu O(i), ponieważ musimy przejść przez całą listę o długości i-1, by dostać się do i-tego węzła.
- Dla porównania, jeżeli zastosujemy tablicę wskaźników do dzieci, do i-tego dziecka dostajemy się w czasie O(1), niezależnie od wartości i.

Reprezentacja drzewa przy wykorzystaniu listy jednokierunkowej

- **W reprezentacji drzew zwanej skrajnie lewy potomek-prawy element siostrzany (ang. left-most-child-right-sibling), w każdym węźle umieszczamy jedynie wskaźniki do skrajnie lewego dziecka; węzeł nie zawiera wskaźników do żadnego ze swoich pozostałych dzieci.**
- **Aby odnaleźć drugi i wszystkie kolejne dzieci węzła n, tworzymy listę jednokierunkowa tych dzieci w której każde dziecko c wskazuje na znajdujące się bezpośrednio po jego prawej stronie dziecko węzła n.**
- **Wskazany węzeł nazywamy prawym elementem siostrzanym węzła c.**

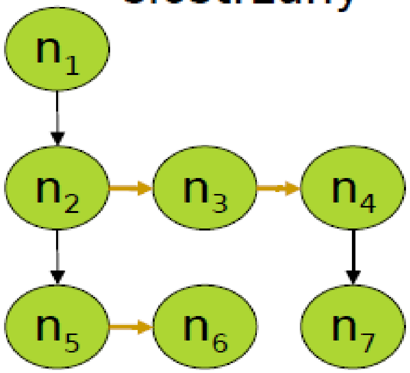
Reprezentacje drzewa

Drzewo złożone z 7 węzłów



info – etykieta
leftmostChild – informacja o węźle
rightSibling – część listy jednokierunkowej dzieci rodzica tego węzła

Reprezentacja skrajnie lewy potomek-prawy element siostrzany



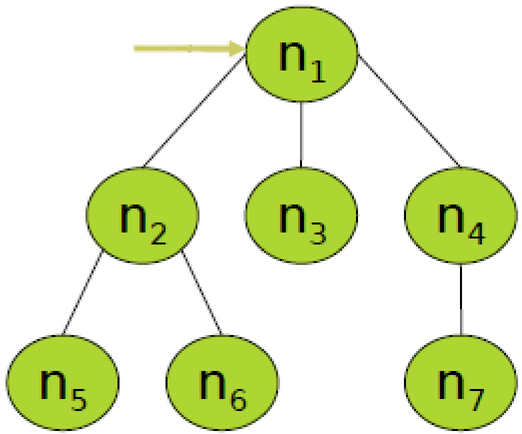

```
typedef struct NODE *pNODE;
struct NODE{
        int info;
        pNODE leftmostChild, rightSibling;
};
```

Reprezentacje drzewa

- **Reprezentacja oparta na tablicy wskaźników** umożliwia nam dostęp do i-tego dziecka dowolnego węzła w czasie O(1). Taka reprezentacja wiąże się jednak ze znacznym marnotrawstwem przestrzeni pamięciowej, jeśli tylko kilka węzłów ma wiele dzieci. W takim wypadku większość wskaźników w tablicy children będzie równa NULL.
- **Reprezentacja skrajnie lewy potomek-prawy element siostrzany** wymaga mniejszej przestrzeni pamięciowej. Nie wymaga również istnienia maksymalnego czynnika rozgałęzienie węzłów. Możemy reprezentować węzły z dowolna wartością tego czynnika, nie modyfikując jednocześnie struktury danych.

Rekurencja w drzewach

- Użyteczność drzew wynika z liczby możliwych operacji rekurencyjnych, które możemy na nich wykonać w naturalny i jasny sposób (chcemy drzewa przeglądać).
- Przeszukiwanie w głąb:
  - Prosta rekurencja zwraca etykiety węzłów w porządku wzdłużnym (ang. pre-order listing), czyli: korzeń, lewe poddrzewo, prawe poddrzewo.
  - Inną powszechnie stosowaną metodą do przeglądania węzłów drzewa jest tzw. przeszukiwanie w porządku wstecznym (ang. post-order listing), czyli lewe poddrzewo, prawe poddrzewo, korzeń.
- Przeszukiwanie w szerz: wszystkie dzieci zaczynając od najbardziej lewego potomka.

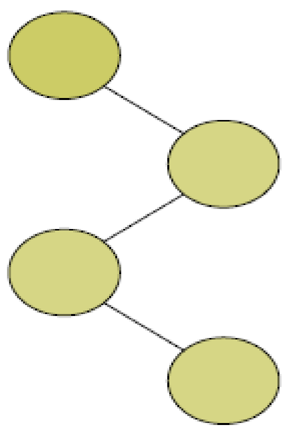


Drzewa binarne

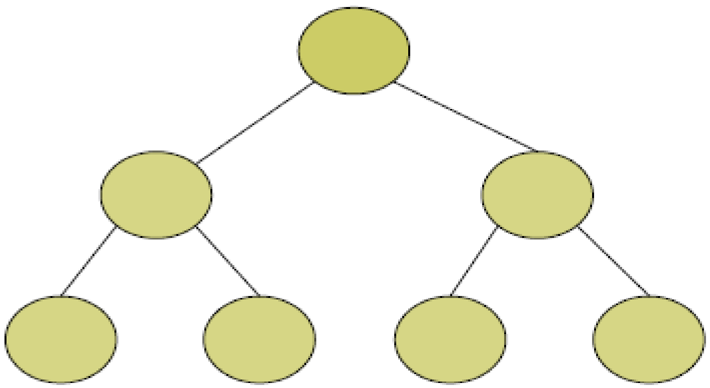
- W drzewie binarnym węzeł może mieć co najwyżej dwoje bezpośrednich potomków.
- Rekurencyjna definicja drzewa binarnego:
  - Podstawa:
    - Drzewo puste jest drzewem binarnym.
  - Indukcja:
    - Jeśli r jest węzłem oraz T_1, T_2 są drzewami binarnymi, istnieje drzewo binarne z korzeniem r, lewym poddrzewem T_1 i prawym poddrzewem T_2. Korzeń drzewa T_1 jest lewym dzieckiem węzła r, chyba że T_1 jest drzewem pustym. Podobnie korzeń drzewa T_2 jest prawym dzieckiem węzła r, chyba że T_2 jest drzewem pustym.
    - Większość terminologii wprowadzonej przy okazji drzew stosuje się oczywiście też do drzew binarnych.
- Różnica: drzewa binarne wymagają rozróżnienia lewego od prawego dziecka, zwykle drzewa tego nie wymagają. Drzewa binarne to NIE są zwykle drzewa, w których węzły mogą mieć co najwyżej dwójkę dzieci.

Drzewa binarne

Zdegenerowane drzewo binarne



Pełne drzewo binarne



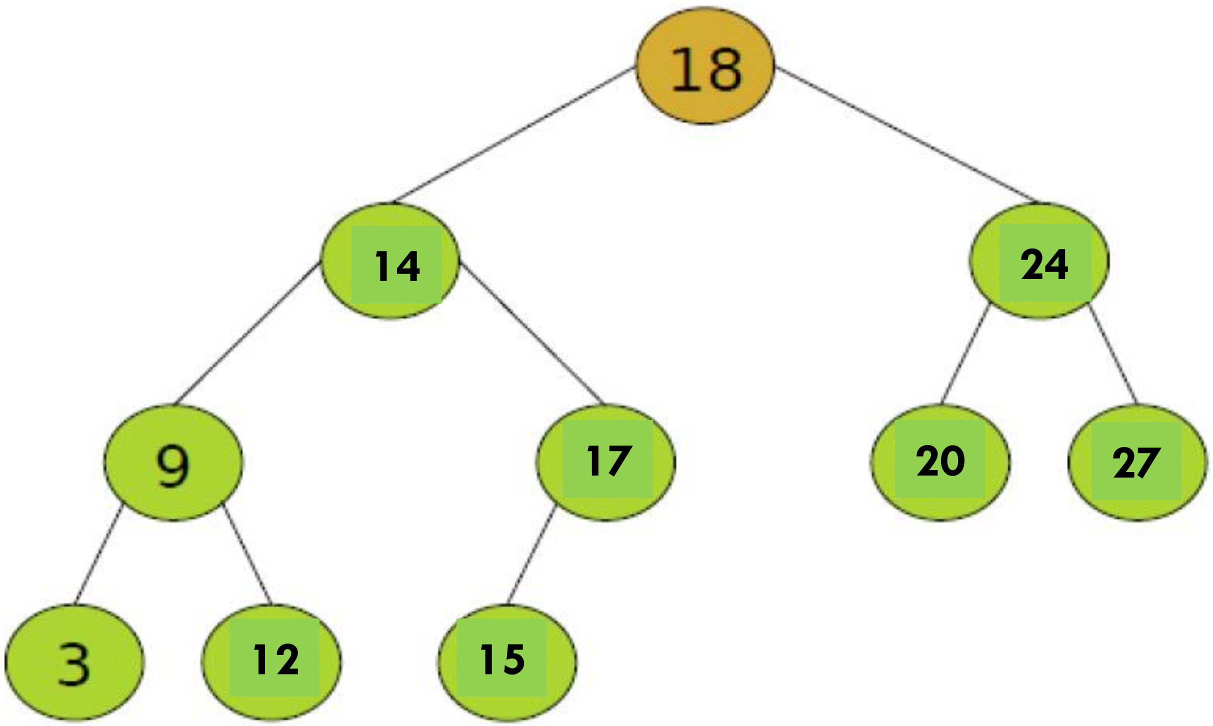
Wysokość drzewa złożonego z k-węzłów to k-1.
Czyli h = O(k).
Operacje insert, delete, find wymagają średnio O(k).

Drzewo o wysokości h ma $k=2^{h+1}-1$ węzłów.
Czyli h = O(log k).
Operacje insert, delete, find wymagają średnio O(log k).

Drzewa przeszukiwania binarnego

- Jest to zaetykietowane drzewo binarne dla którego etykiety należą do zbioru w którym możliwe jest zdefiniowanie relacji mniejszości.
- Dla każdego węzła x spełnione są następujące własność:
  - wszystkie węzły w lewym poddrzewie maja etykiety mniejsze od etykiety węzła x
  - wszystkie w prawym poddrzewie maja etykiety większe od etykiety węzła x.

Drzewa przeszukiwania binarnego



Drzewa przeszukiwania binarnego

- Wyszukiwanie elementu:
  - Podstawa:
    - Jeśli drzewo T jest puste, to na pewno nie zawiera elementu x.
    - Jeśli T nie jest puste i szukana wartość x znajduje się w korzeniu, drzewo zawiera x.
  - Indukcja:
    - Jeśli T nie jest puste, ale nie zawiera szukanego elementu x w korzeniu, niech y będzie elementem w korzeniu drzewa T.
    - Jeśli x<y, szukamy wartości x tylko w lewym poddrzewie korzenia y.
    - Jeśli x>y, szukamy wartości x tylko w prawym poddrzewie korzenia y.

Własność drzewa przeszukiwania binarnego gwarantuje, że szukanej wartości x na pewno nie ma w poddrzewie, którego nie przeszukujemy

Drzewa przeszukiwania binarnego

- **Wstawianie elementu:**
  - **Podstawa:**
    - Jeśli drzewo T jest drzewem pustym, zastępujemy T drzewem składającym się z pojedynczego węzła zawierającego element x.
    - Jeśli drzewo T nie jest puste oraz jego korzeń zawiera element x, to x znajduje się już w drzewie i nie wykonujemy żadnych dodatkowych kroków.
  - **Indukcja:**
    - Jeśli T nie jest puste i nie zawiera elementu x w swoim korzeniu, niech y będzie elementem w korzeniu drzewa T.
    - Jeśli $x<y$, wstawiamy wartość x do lewego poddrzewa T.
    - Jeśli $x>y$, wstawiamy wartość x do prawego poddrzewa T.

Drzewa przeszukiwania binarnego

Usuwanie elementu:

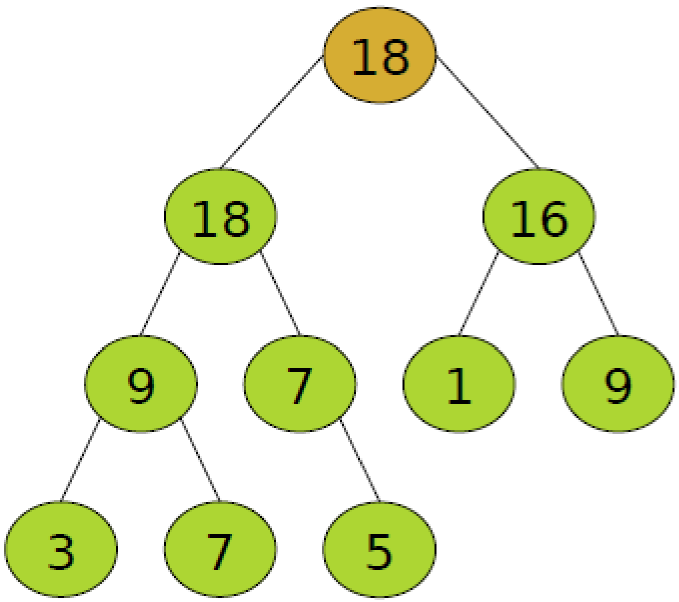
- Usuwanie elementu x z drzewa przeszukiwania binarnego jest zadaniem nieco bardziej skomplikowanym od znajdowania czy wstawiania danego elementu. Musimy zachować własność drzewa przeszukiwania binarnego.
- Lokalizujemy x, oznaczmy węzeł w którym się on znajduje poprzez v.
  - Jeśli drzewo nie zawiera x to nie robimy nic.
  - Jeżeli v jest liściem to go usuwamy.
  - Jeśli v jest wewnętrznym węzłem i węzeł ten ma tylko jedno dziecko, przypisujemy węzłowi v wartość dziecka v, a następnie usuwamy dziecko v. (W ten sposób że dziecko dziecka v, staje się dzieckiem v, a rodzicem dziecka dziecka v staje się v).
  - Jeżeli węzeł v ma dwoje dzieci, oznaczmy poprzez y najmniejszą wartość w prawym poddrzewie v. Następnie przypisujemy węzłowi v wartość y, i usuwamy y z prawego poddrzewa v.

Drzewo przeszukiwania binarnego – efektywna implementacja słownika

- Często stosowaną w programach komputerowych modelem danych jest zbiór, na którym chcemy wykonywać operacje:
  - wstawianie nowych elementów do zbioru (ang. insert)
  - usuwanie elementów ze zbioru (ang. delete)
  - wyszukiwanie jakiegoś elementu w celu sprawdzenia, czy znajduje się w danym zbiorze (ang. find)
- Taki zbiór będziemy nazywać słownikiem (niezależnie od tego jakie elementy zawiera). Drzewo przeszukiwania binarnego umożliwia stosunkowo efektywną implementację słownika.
- Czas wykonania każdej z operacji na słowniku reprezentowanym przez drzewo przeszukiwania binarnego złożone z n węzłów jest proporcjonalny do wysokości tego drzewa h.

Drzewa binarne częściowo uporządkowane

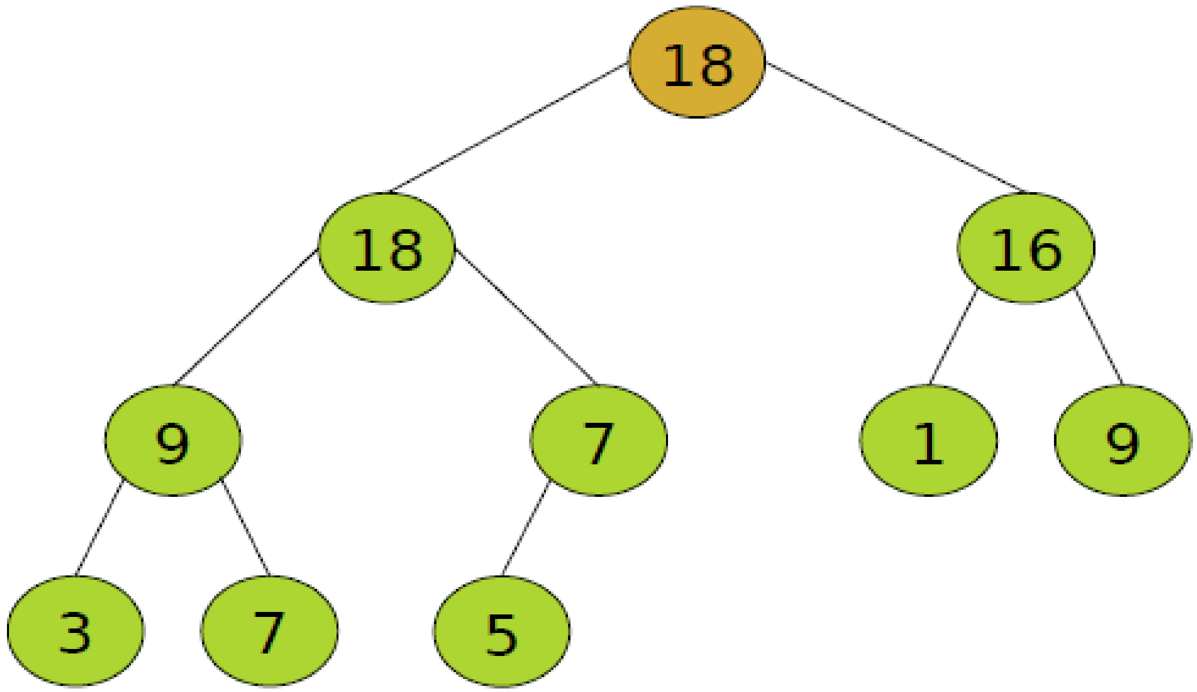
- Jest to zaetykietowane drzewo binarne o następujących własnościach:
  - Etykietami węzłów są elementy z przypisanymi priorytetami; priorytet może być wartością elementu lub przynajmniej jednego z jego komponentów.
  - Element przechowywany w węźle musi mieć co najmniej tak duży priorytet jak element znajdujący się w dzieciach tego węzła. Element znajdujący się w korzeniu dowolnego poddrzewa jest wiec największym elementem tego poddrzewa.



Zrównoważone drzewa częściowo uporządkowane i kopce

- **Mówimy że drzewo uporządkowane jest zrównoważone (ang. balanced), jeśli na wszystkich poziomach poza najniższym zawiera wszystkie możliwe węzły oraz liście na najniższym poziomie są ułożone od lewej strony. Spełnienie tego warunku oznacza, że jeśli drzewo składa się z n węzłów, to żadna ścieżka od korzenia do któregokolwiek z tych węzłów nie jest dłuższa niż $\log_2 n$.**
- **Zrównoważone drzewa częściowo uporządkowane można implementować za pomocą tablicowej struktury danych zwanej kopcem (ang. heap), która umożliwia szybką i zwięzłą implementację kolejek priorytetowych. Kopiec jest to po prostu tablica A, której sposób indeksowania reprezentujemy w specyficzny sposób. Zapisuje się kolejne poziomy, zawsze porządkując od lewej do prawej.**

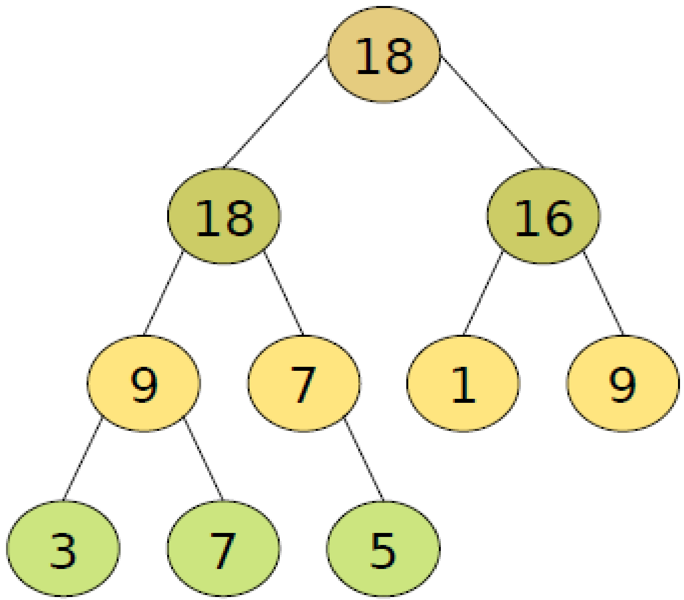
Zrównoważone drzewa częściowo uporządkowane i kopce



Kopiec dla zrównoważonego częściowo uporządkowanego drzewa.

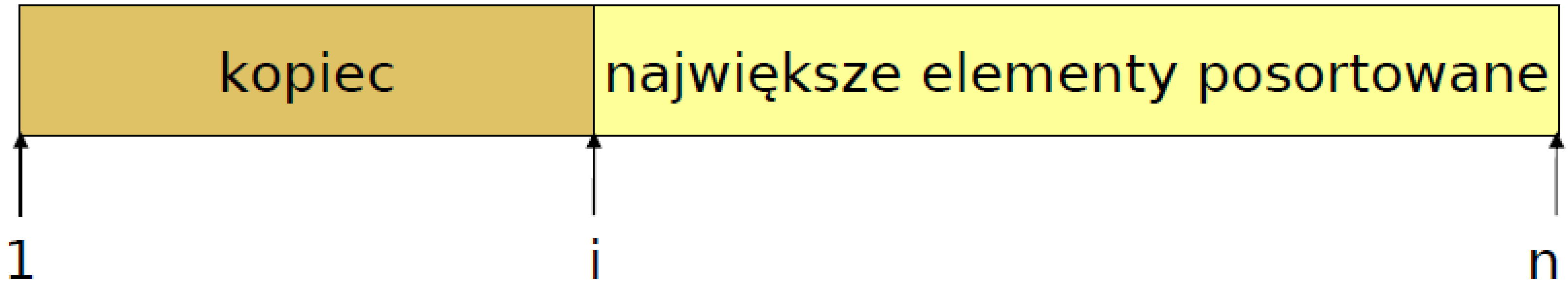
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
18	18	16	9	7	1	9	3	7	5

- **Rozpoczynamy od korzenia A[1]; nie wykorzystujemy A[0].**
- **Po korzeniu zapisujemy kolejne poziomy, w każdym poziomie węzły porządkujemy od lewej do prawej.**
- **Lewe dziecko korzenia znajduje się w A[2]; prawe dziecko korzenia umieszczamy w A[3].**
- **W ogólności, lewe dziecko węzła zapisane w A[i] znajduje się w A[2i], prawe dziecko tego samego węzła znajduje się w A[2i+1], jeśli oczywiście te dzieci istnieją w drzewie uporządkowanym.**
- **Taka reprezentacja jest możliwa dzięki własnościom drzewa zrównoważonego.**
- **Z własności drzewa częściowo uporządkowanego wynika, że jeśli A[i] ma dwójkę dzieci, to A[i] jest co najmniej tak duże, jak A[2i] i A[2i+1], oraz jeśli A[i] ma jedno dziecko, to A[i] nie jest mniejsze niż A[2i].**



Sortowanie przez kopcowanie

- **Za pomocą tego algorytmu sortujemy tablice A[1,…n] w dwóch etapach:**
  - **algorytm nadaje tablicy A własność drzewa częściowo uporządkowanego**
  - **wielokrotnie wybiera największy z pozostałych elementów z kopca aż do momentu, w którym na kopcu znajduje się tylko jeden (najmniejszy) element co oznacza że tablica jest posortowana.**

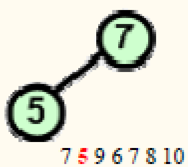
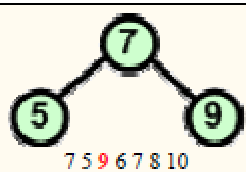
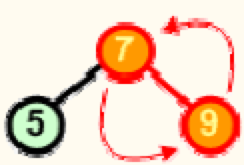


- **Wykonanie operacji sortowania przez kopcowanie wymaga czasu O(nlogn). Dla porównania sortowanie przez wybieranie wymaga czasu $O(n^2)$.**

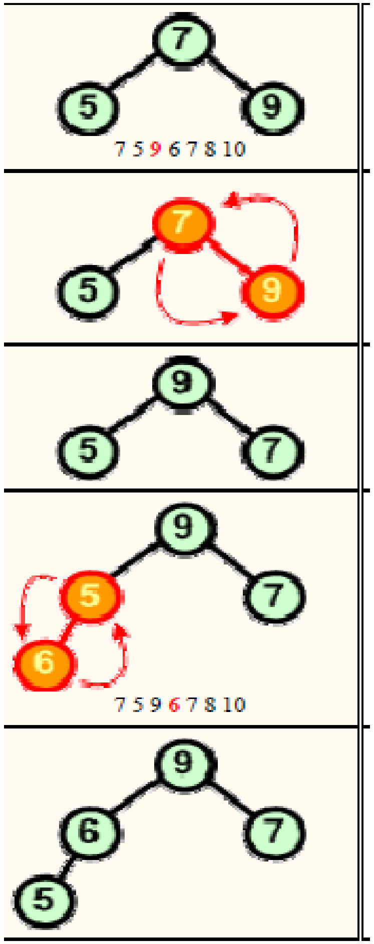
Kopiec : tworzenie

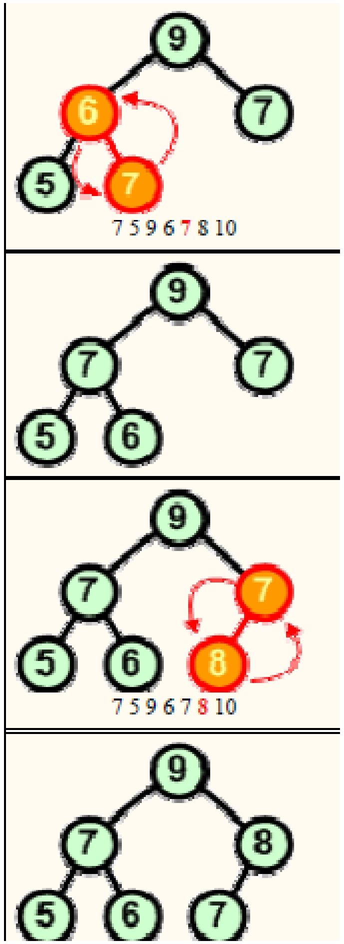
Kopiec jest drzewem binarnym, w którym wszystkie węzły spełniają następujący warunek (zwany warunkiem kopca) : węzeł nadrzędny jest większy lub równy węzłom potomnym (w porządku malejącym relacja jest odwrotna - mniejszy lub równy).

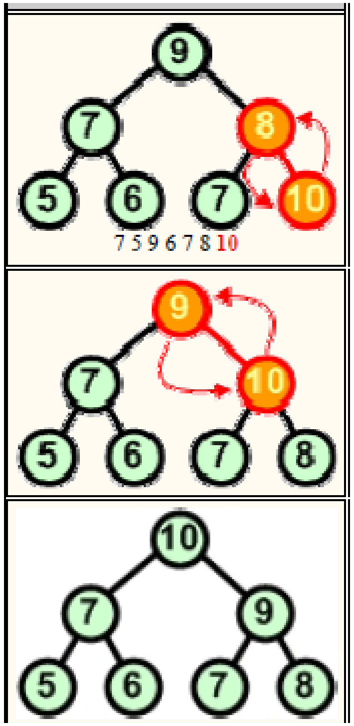
Przykład : Skonstruować kopiec z elementów zbioru {7 5 9 6 7 8 10}

Operacja	Opis
	Budowę kopca rozpoczynamy od pierwszego elementu zbioru, który staje się korzeniem.
	Do korzenia dołączamy następny element. Warunek kopca jest zachowany.
	Dodajemy kolejny element ze zbioru.
	Po dodaniu elementu 9 warunek kopca przestaje być spełniony. Musimy go przywrócić. W tym celu za nowy węzeł nadrzędny wybieramy nowo dodany węzeł. Poprzedni węzeł nadrzędny wędruje w miejsce węzła dodanego - zamieniamy węzły 7 i 9 miejscami.

Kopiec

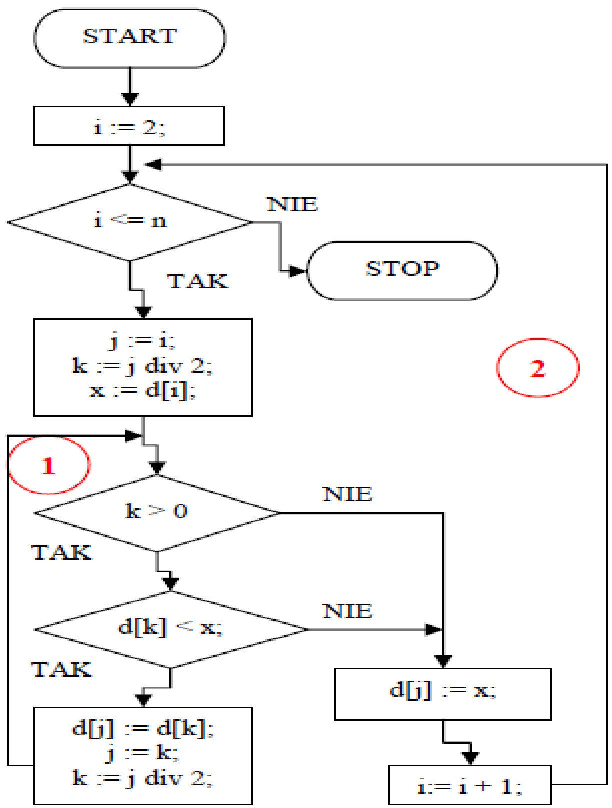






Kopiec

Schemat blokowy



Złożoność obliczeniowa O(n log(n))

Algorytm tworzy kopiec w tym samym zbiorze wejściowym d[ ]. Nie wymaga zatem dodatkowych struktur danych i ma złożoność pamięciową klasy Θ(n).
Pętla nr 1 wyznacza kolejne elementy wstawiane do kopca. Pierwszy element pomijamy, ponieważ zostałby i tak na swoim miejscu. Dlatego pętla rozpoczyna wstawianie od elementu nr 2.
Wewnątrz pętli nr 1 inicjujemy kilka zmiennych:
j - pozycja wstawianego elementu (liścia)
k - pozycja elementu nadrzędnego (przodka)
x - zapamiętuje wstawiany element

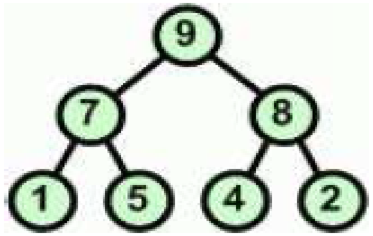
Następnie rozpoczynamy pętlę warunkową nr 2, której zadaniem jest znalezienie w kopcu miejsca do wstawienia zapamiętanego elementu w zmiennej x. Pętla ta wykonuje się do momentu osiągnięcia korzenia kopca (k = 0) lub znalezienia przodka większego od zapamiętanego elementu. Wewnątrz pętli przesuwamy przodka na miejsce potomka, aby zachować warunek kopca, a następnie przesuwamy pozycję j na pozycję zajmowaną wcześniej przez przodka. Pozycja k staje się pozycją nowego przodka i pętla się kontynuuje. Po jej zakończeniu w zmiennej j znajduje się numer pozycji w zbiorze d[ ], na której należy umieścić element w x.
Po zakończeniu pętli nr 1 w zbiorze zostaje utworzona struktura kopca.

Kopiec: rozbieranie

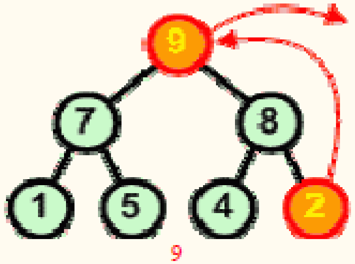
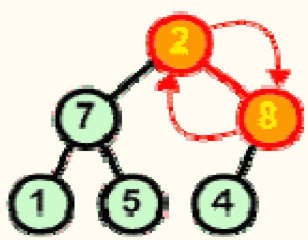
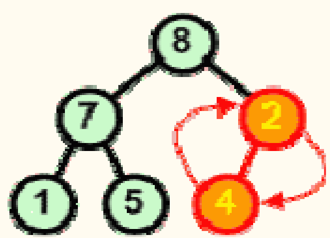
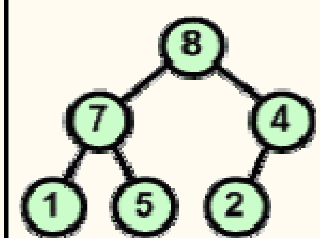
Zasady rozbioru kopca :

1. Zamień miejscami korzeń z ostatnim liściem, który wyłącz ze struktury kopca. Elementem pobieranym z kopca jest zawsze jego korzeń, czyli element największy.
2. Jeśli jest to konieczne, przywróć warunek kopca idąc od korzenia w dół.
3. Kontynuuj od kroku 1, aż kopiec będzie pusty.

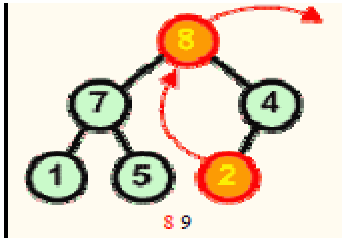
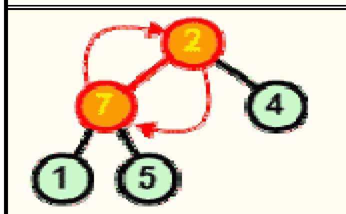
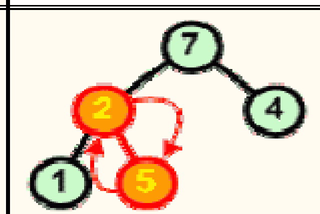
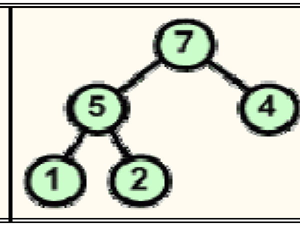
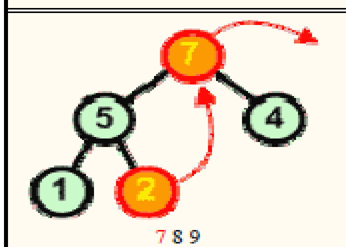
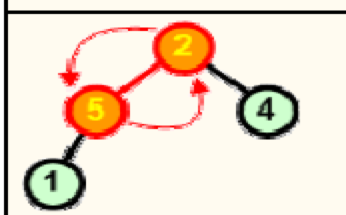
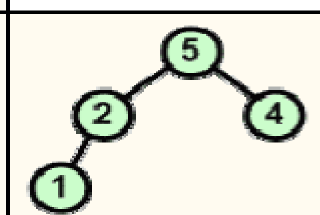
Przykład : Rozebrać kopiec



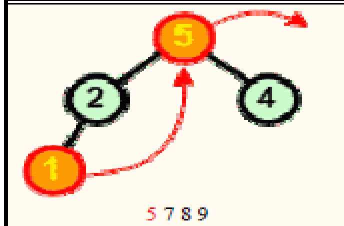
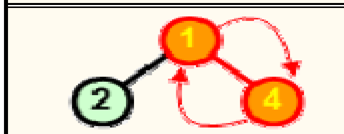
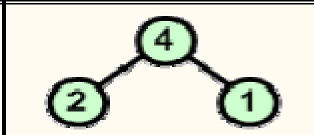
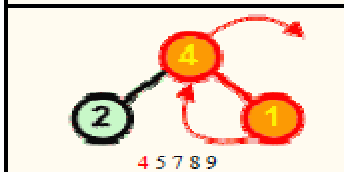
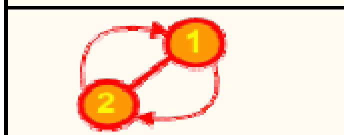
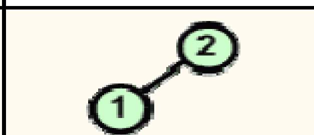
Kopiec: rozbieranie

<table>
<tr><th>Operacja</th><th colspan="3">Opis</th></tr>
<tr><td></td><td colspan="3">Rozbiór kopca rozpoczynamy od korzenia, który usuwamy ze struktury kopca. W miejsce korzenia wstawiamy ostatni liść.</td></tr>
<tr><td></td><td></td><td></td><td>Poprzednia operacja zaburzyła strukturę kopca. Idziemy zatem od korzenia w dół struktury przywracając warunek kopca - przodek większy lub równy od swoich potomków. Praktycznie polega to na zamianie przodka z największym potomkiem. Operację kontynuujemy dotąd, aż natrafimy na węzły spełniające warunek kopca.</td></tr>
</table>

Kopiec: rozbieranie

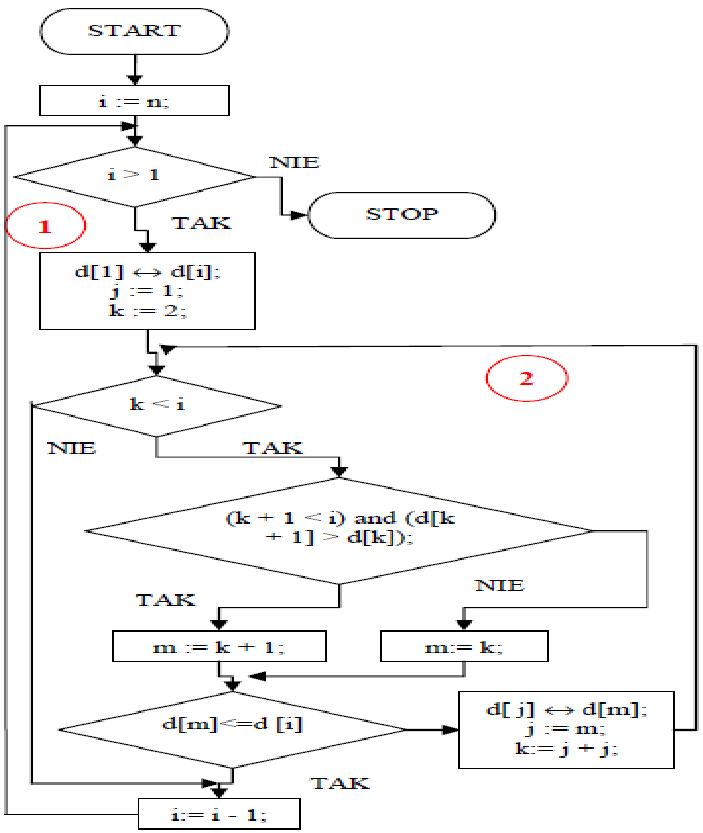
<table>
<tr><td>
</td><td colspan="3">Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem</td></tr>
<tr><td>
</td><td>
</td><td>
</td><td>W nowym kopcu przywracamy warunek kopca.</td></tr>
<tr><td>
</td><td colspan="3">Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem</td></tr>
<tr><td>
</td><td>
</td><td colspan="2">W nowym kopcu przywracamy warunek kopca. W tym przypadku już po pierwszej wymianie węzłów warunek koca jest zachowany w całej strukturze.</td></tr>
</table>

Kopiec: rozbieranie

	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem	
		Przywracamy warunek kopca w strukturze.
	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem	
		Przywracamy warunek kopca w strukturze.
	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem.	
	Po wykonaniu poprzedniej operacji usunięcia w kopcu pozostał tylko jeden element - usuwamy go. Zwróć uwagę, iż usunięte z kopca elementy tworzą ciąg uporządkowany.	

Kopiec: rozbieranie

Schemat blokowy



Rozbiór kopca wykonywany jest w dwóch zagnieżdżonych pętlach. Pętla nr 1 zamienia miejscami kolejne liście ze spodu drzewa z korzeniem. Zadaniem pętli nr 2 jest przywrócenie w strukturze warunku kopca.

Złożoność obliczeniowa

O(n log(n))

Sortowanie przez kopcowanie

Krok 1 : Tworz_Kopiec
Krok 2 : Rozbierz_Kopiec
Krok 3 : Zakończ algorytm

Cechy Algorytmu Sortowania Przez Kopcowanie	
klasa złożoności obliczeniowej optymistyczna	Θ (n log n)
klasa złożoności obliczeniowej typowa	
klasa złożoności obliczeniowej pesymistyczna	
Sortowanie w miejscu	TAK
Stabilność	NIE

Ponieważ sortowanie przez kopcowanie składa się z dwóch następujących bezpośrednio po sobie operacji o klasie czasowej złożoności obliczeniowej Θ(n log n), to dla całego algorytmu klasa złożoności również będzie wynosić Θ (n log n).

Algorytmy sortujące

Nazwa algorytmu sortującego	Klasa złożoności			Stabilność	Sortowanie w miejscu	Zalecane?
	optymistyczna	typowa	pesymistyczna			
Zwariowane	$\Theta(1)$	$\Theta(n*n!)$	$\Theta(\infty)$	NIE	TAK	NIE!!!
Naiwne	$\Theta(n) \ldots \Theta(n^2)$	$\Theta(n^3)$	$\Theta(n^3)$	TAK	TAK	NIE
Bąbelkowe wersja 1	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	NIE
Bąbelkowe wersja 2	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	TAK
Przez wybór	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	NIE	TAK	TAK/NIE
Przez wstawianie	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	TAK
Metodą Shella	$\Theta(n^{1,14})$	$\Theta(n^{1,15})$	$\Theta(n^{1,15})$	NIE	TAK	TAK
Przez łączenie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	TAK	NIE	TAK
Przez kopcowanie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	NIE	TAK	TAK
Szybkie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	NIE	TAK	TAK
Kubełkowe wersja I	$\Theta(m + n)$	$\Theta(m + n)$	$\Theta(m + n)$	NIE	NIE	TAK/NIE
Radix Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	TAK	NIE	TAK

Kolejka priorytetowa

- Inny typ danych to zbiór elementów, z których każdy jest związany z określonym priorytetem. Przykładowo, elementy mogą być strukturami, zaś priorytet może być wartością jednego z pól takiej struktury. Chcemy wykonywać operacje:
  - wstawianie nowych elementów do zbioru (ang. insert)
  - znalezienie i usuniecie ze zbioru elementu o najwyższym priorytecie (ang. deletemax)
- Taki zbiór będziemy nazywać kolejką priorytetowa (niezależnie od tego jakie elementy zawiera).
- Drzewo binarne częściowo uporządkowane umożliwia stosunkowo efektywną implementację kolejki priorytetowej.
- Efektywna (używając kopca) tzn. O(nlog n).

Operacje kolejki priorytetowej na kopcu

Reprezentujemy kopiec za pomocą globalnej tablicy liczb całkowitych A[1,...,MAX]. Przypuśćmy, że mamy kopiec złożony z n-1 elementów, który spełnia własność drzewa częściowo uporządkowanego.

- Operacja insert:
  - Dodajemy n-ty element w A[n].
  - Własność drzewa uporządkowanego jest nadal spełniona we wszystkich elementach tablicy, poza (być może) elementem A[n] i jego rodzicem.
  - Jeśli element A[n] jest większy od elementu A[n/2], czyli jego rodzica, musimy wymienić te elementy ze sobą (ta operacje nazywamy sortowanie bąbelkowe w górę (ang. bubbleUp).
  - Może teraz zaistnieć konflikt z własnością drzewa częściowo uporządkowanego pomiędzy elementem A[n/2] i jego rodzicem. Sprawdzamy i ewentualnie wymieniamy ich pozycje.
  - Itd.
- Operacja deletemax:
  - A[1] przypisujemy wartość -∞ , po czym wykorzystujemy analogiczną procedurę co powyżej, czyli: sortowanie bąbelkowe w dół (ang. bubbleDown) niech k oznacza pozycję w tablicy liścia do którego zejdzie wartość -∞. A[k] przypiszmy wartość A[n]. Po czym wykonajmy procedure sortowania bąbelkowego w górę. Kopiec będzie teraz reprezentowany przez tablicę A[1, …, n-1].

Wykonywanie operacji insert i deletemax wymaga czasu O(log n).

Poziomy implementacji

- **Dwa abstrakcyjne typy danych:**

  słownik i kolejka priorytetowa

- **Omówiliśmy dwie różne abstrakcyjne implementacje i wybraliśmy konkretne struktury danych dla każdej z tych abstrakcyjnych implementacji.**

Abstrakcyjny typ danych	Abstrakcyjna implementacja	Struktura danych
Słownik	Drzewo przeszukiwania binarnego	Struktura lewe dziecko – prawe dziecko
Kolejka priorytetowa	Zrównoważone drzewo częściowo uporządkowane	Kopiec

Podsumowanie

- Ważnym modelem danych reprezentującym informacje hierarchiczne są drzewa.
- Do implementowania drzew możemy wykorzystać wiele różnych struktur danych (także takich) które wymagają połączenia tablic ze wskaźnikami. Wybór struktury danych zależy od operacji wykonywanych na drzewie.
- Dwiema najważniejszymi reprezentacjami węzłów drzewa są skrajnie lewy potomek-prawy element siostrzany oraz tree (tablica wskaźników do dzieci).
- Drzewa nadają się doskonale do stosowania na nich algorytmów i dowodów rekurencyjnych.
- Drzewo binarne jest jednym z wariantów modelu drzewa, w którym każdy węzeł ma (opcjonalne) lewe i prawe dziecko.
- Drzewo przeszukiwania binarnego jest zaetykietowanym drzewem binarnym, które spełnia „własność drzewa przeszukiwania binarnego”.

Podsumowanie

- **Będący abstrakcyjnym typem danych, słownik jest zbiorem, na którym można wykonywać operacje insert, delete, find. Efektywna implementacja słownika to drzewo przeszukiwania binarnego.**
- **Innym abstrakcyjnym typem danych jest kolejka priorytetowa, czyli zbiór na którym możemy wykonywać operacje insert i deletemax. Efektywna implementacja to kopiec.**
  - **Drzewo częściowo uporządkowane jest zaetykietowanym drzewem binarnym spełniającym warunek, że żadna etykieta w żadnym węźle nie jest mniejsza od etykiety żadnego z jego dzieci.**
  - **Zrównoważone drzewo częściowo uporządkowane (węzły całkowicie wypełniają wszystkie poziomy od korzenia do najniższego, w którym zajmowane są tylko skrajnie lewe pozycje) możemy implementować za pomocą kopca. Struktura ta umożliwia implementację operacji na kolejce priorytetowej wykonywanych w czasie O(log n) oraz algorytmu sortującego, zwanego sortowaniem przez kopcowanie działającego w czasie O(n log n) .**

Pytania do egzaminu

1) **Jakie operacje wykonujemy na zbiorach?**
2) **Wytłumacz dlaczego implementacje danych typu „zbiór” przy pomocy list jednokierunkowych znacząco skraca czas wykonywania operacji „suma zbiorów”?**
3) **Porównaj złożoność operacji dla implementacji zbioru na liście jednokierunkowej, wektorze własnym i tablicy mieszającej.**
4) **Jaki znasz struktury danych używane do reprezentacji drzew?**
5) **Jaka działa algorytm przeszukiwania drzewa w głąb? Jak działa algorytm przeszukiwania drzewa w szerz.**
6) **Co to jest drzewo binarne? Jaka jest zależność max i min wysokość drzewa od ilości węzłów.**
7) **Co to jest drzewo przeszukiwania binarnego?**
8) **Jak są realizowane podstawowe operacje operacje dla drzew przeszukiwania binarnego? Jakie to są operacje?**
9) **Czym charakteryzuje się zrównoważone drzewo częściowo uporządkowane?**
10) **Na czym polega sortowanie przez kopcowanie?**
11) **W jaki sposób wykorzystuje się kopiec dla implementacji kolejki priorytetowej.**