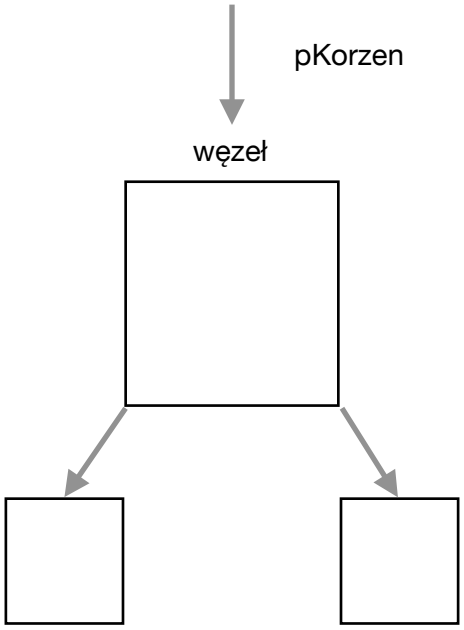
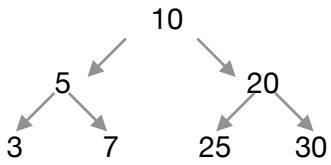
Drzewo poszukiwań binarnych

Każdy element jest nazywany węzłem. Są dwa wskaźniki - na lewy i prawy element.



Elementy większe po stronie prawej, mniejsze po lewej. Przy każdym węźle odrzucamy połowę przypadków, np. szukając 30 odrzucimy lewą stronę już przy pierwszych węzłach.



if (wartosc < pRoot -> wartosc) - jeśli wartość jest mniejsza niż to, co przechowuje pRoot, idź w lewo.

Wypisywanie: najpierw wypisujemy całą lewą stronę, potem pierwszy element, potem prawą stronę. Wypisanie jest rekurencyjne. W taki sposób wypisane elementy będą posortowane.

Można też wypisywać drzewo z „wcięciem” - setw (wciecie * MNOZNIK) - bardziej przypomina drzewo.

Należy usunąć elementy z pamięci, gdy skończy się korzystać z drzewa (jak z listą). USUWAĆ TEŻ WSKAŹNIKI! (koperta z Akademicką 16, a budynku nie ma) pRoot -> nullptr;

Iteracyjne tworzenie i usuwanie drzewa: trzeba stworzyć dodatkowy wskaźnik, aby nie poruszać się pRootem.

Można też szukać elementów w drzewie rekurencyjnie lub iteracyjnie. Zwracamy adres, nie sam element.

Własny operator wypisania

strumień wyjściowy (ostream) & operator << (ostream & nazwa strumienia, to co zwracamy)

Strumień należy zwrócić.

Funkcje operujące na drzewie

- **LICZENIE WĘZŁÓW (ELEMENTÓW)**
  1 (pierwszy korzeń) + liczenie węzłów z lewej + liczenie węzłów z prawej
- **SUMOWANIE WSZYSTKICH WARTOŚCI W DRZEWIE**
  inicjować sumę domyślną wartością { }, nie zerem (nie znamy typu, który będzie przechowywało drzewo)
  zwrócić wartości korzenia, lewej strony i prawej strony
- **SUMOWANIE LIŚCI (ELEMENTÓW, KTÓRE NIE MAJĄ JUŻ ŻADNYCH POTOMKÓW)**
  jeśli nie ma wskaźnika na lewy lub prawy, zwróć liść
  ufikuśnienie: operator trójargumentowy (jedna linijka)
- **MAKSYMALNA WYSOKOŚĆ DRZEWA**
  liczba węzłów od węzła korzeniowego do najbardziej odległego liścia
  sprawdzić strony (która jest „wyższa") rekurencyjnie
  zwracamy obliczoną wysokość + 1 (pierwszy korzeń) w jednej linijce
  return std::max <wyliczona wartość> + 1;
- **MINIMALNA I MAKSYMALNA WARTOŚĆ W DRZEWIE**
  funkcja zwraca wskaźnik
  wezel * minimalny / maksymalny (wezel * pRoot)
- **SZUKANIE WĘZŁÓW RODZICIELSKICH**
  jak nie ma potomka, to nie ma rodzica (nie ma rodzica osoby nieistniejącej)
  jeśli wskaźniki na lewy/prawy są wskaźnikami na potomka jednocześnie, to znaleźliśmy potomka
  jeśli w potomku jest wartość mniejsza - idziemy lewo, większa - idziemy prawo
  wywołanie funkcji odpowiednio dla lewej i prawej strony

Przechodzenie przez drzewo

- **WGŁĄB (DO KOŃCA W LEWO/PRAWO)**
  podejście już używane w poprzednich funkcjach (przejście lewy/prawy)
- **WSZERZ (PIĘTRAMI)**
  pamiętamy poprzednie węzły
  posługujemy się kolejkami bibliotecznymi (fifo)
  std::deque <wezel *> kolejka;
  kolejka.push_front (pRoot)
  można powyższe działania zrobić też na ręcznie implementowanych listach
  „wyciągamy" adresy i sprawdzamy potomków -> „wyciągamy" kolejne adresy
  adresy pobieramy i wyrzucamy za pomocą push_front i pop_back
  aby wypisywać piętrami pobieramy lewo prawo lewo prawo… poprzez zapętlenie rekurencją
  pusta kolejka oznacza przejście przez całe drzewo

Kontenery

VECTOR
- jak zwykła tablica, elementy przechowywane jeden po drugim
- można łatwo posortować
- można indeksować od 0 do size-1

UWAGA: begin wskazuje na pierwszy element, ale end wskazuje **za** ostatni element

LISTY (JAKO TYP LISTA)
- nie można łatwo zwrócić rozmiaru
- elementy nie są w pamięci obok siebie (jak w vectorze)

MAPA
- można dowolnie indeksować
- wymaga podania dwóch typów - indeksu i przechowywanej wartości
- łatwo można np. policzyć występowanie danych słów w pliku
- można używać do tablic wielowymiarowych

MAPA NIEUPORZĄDKOWANA
- działa szybciej
- elementy nie są uporządkowane
- dostęp do dowolnego elementu jest niezależny od liczby elementów