Priority Queues

L.EIC

Algoritmos e Estruturas de Dados

2023/2024

P Diniz, AP Rocha,
A Costa, B Leite, F Ramos, J Pires, PH Diniz, V Silva

Priority Queue

Priority queue is a collection where

- every element has some priority value associated with it.
- an element with high priority is dequeued before an element with low priority.

A priority queue allows, at least, the following operations on a set of comparable values:

- insert an element
- delete the element with highest priority
- find the element with highest priority

- Implementation:
  - linked lists
  - binary search trees
  - binary heaps

Priority Queue

- Implementation using linked list

  unordered linked list:

  – insert an element: complexity $O(1)$

  – delete the element with highest priority: complexity $O(n)$

    - complexity $O(n)$ to find the element, and $O(1)$ to remove it

  – find the element with highest priority: complexity $O(n)$

  ordered linked list:

  – insert an element: complexity $O(n)$

  – delete the element with highest priority: complexity $O(1)$

  – find the element with highest priority: complexity $O(1)$

what is the best alternative?

Priority Queue

- Implementation using binary search trees

Binary Search Trees:

– insert an element: complexity $O(\log n)$

– delete the element with highest priority: complexity $O(\log n)$

– find the element with highest priority: complexity $O(\log n)$

* complexity $O(\log n)$, average case in binary search trees

* complexity $O(\log n)$, worst case, if balanced binary search trees

Priority Queue

- Implementation using binary heaps

  First, let´s see the definition of Complete Binary Trees: all levels are completely filled, with the possible exception of the last one which will be filled from the left. So:

  – a complete binary tree is balanced

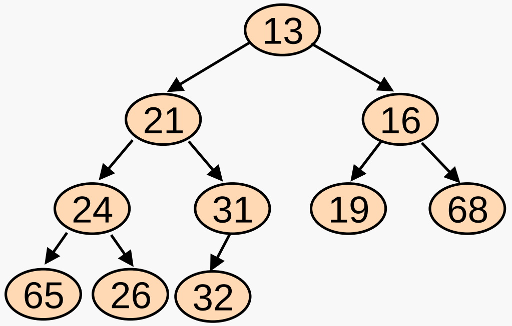
  – a complete binary tree can be *represented in a vector* (less space)

Binary Heap:

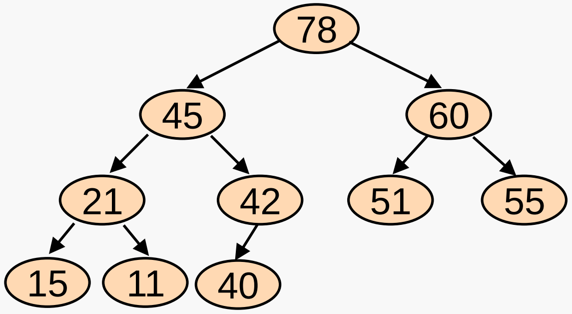
– can be visualized as a complete binary tree

– represented in a vector (tree visit by level)

– for all nodes, except the root, the value of the parent is less/higher than or equal to the value of the node



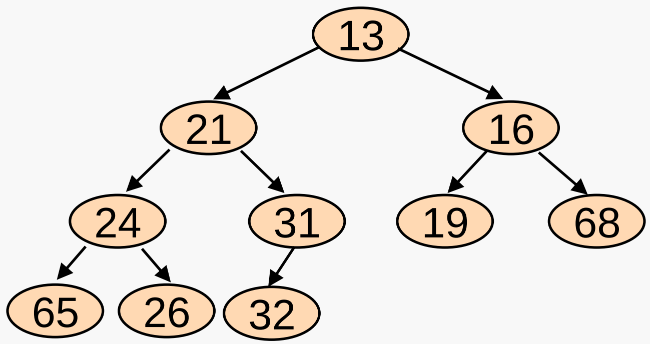
min-heap



max-heap

Binary Heap

consider, as example, the ***min-heap***



array representation:

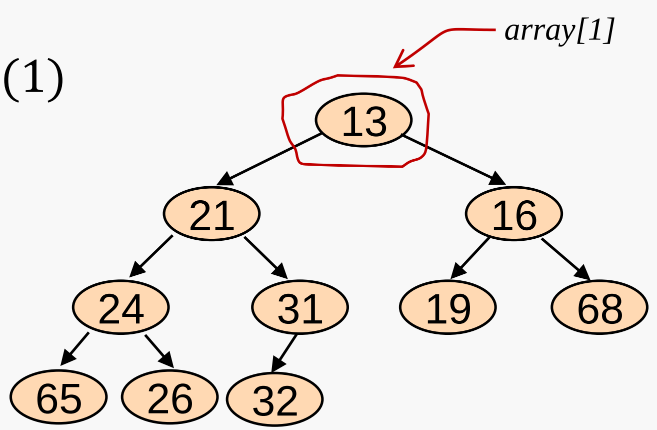
[13, 21, 16, 24, 31, 19, 68, 65, 26, 32] (if starts at index 0)

$i \longrightarrow$ left child: $2 \times i + 1$

right child: $2 \times i + 2$

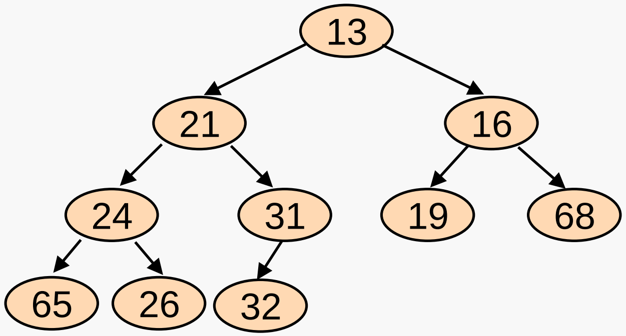
$i \longrightarrow$ parent: $(i - 1)/2$

- Find the element with highest priority: $O(1)$

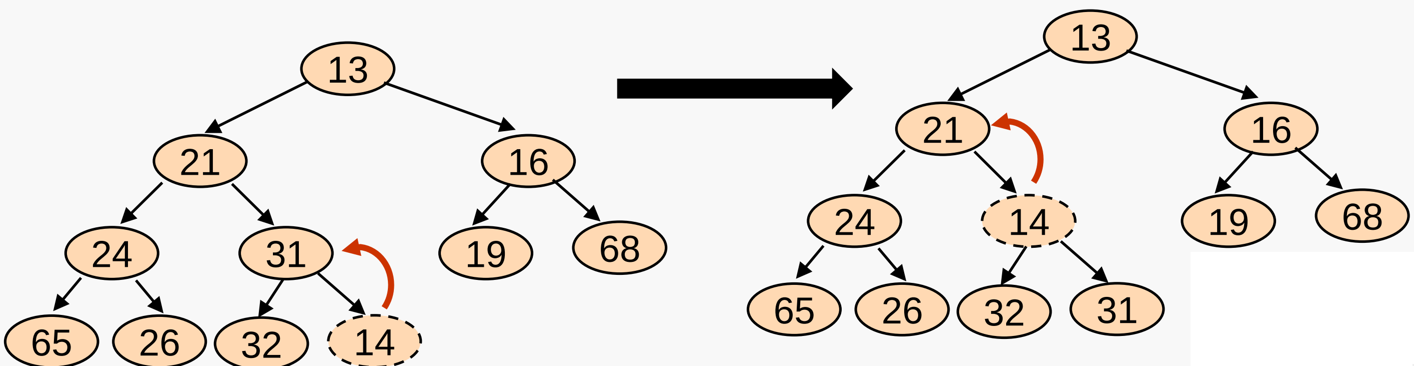


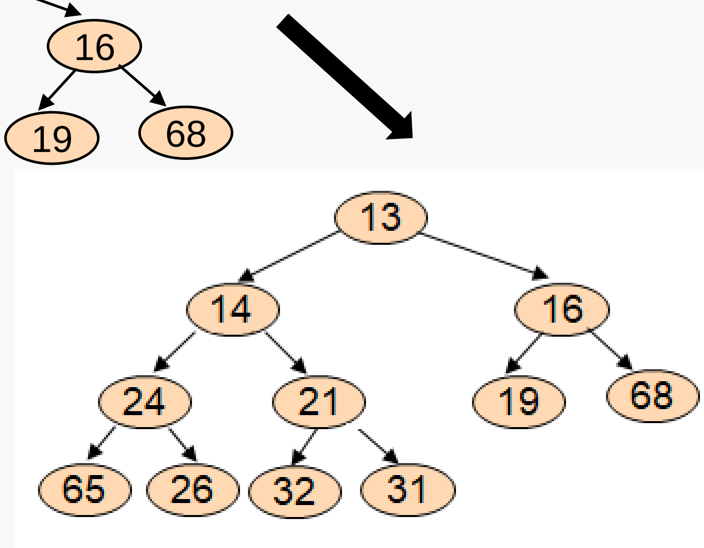
Binary Heap

- Insert an element: $O(\log n)$
  - insert element X in first free position
  - as long as the order is not respected: swap element X and its parent



example: insert 14

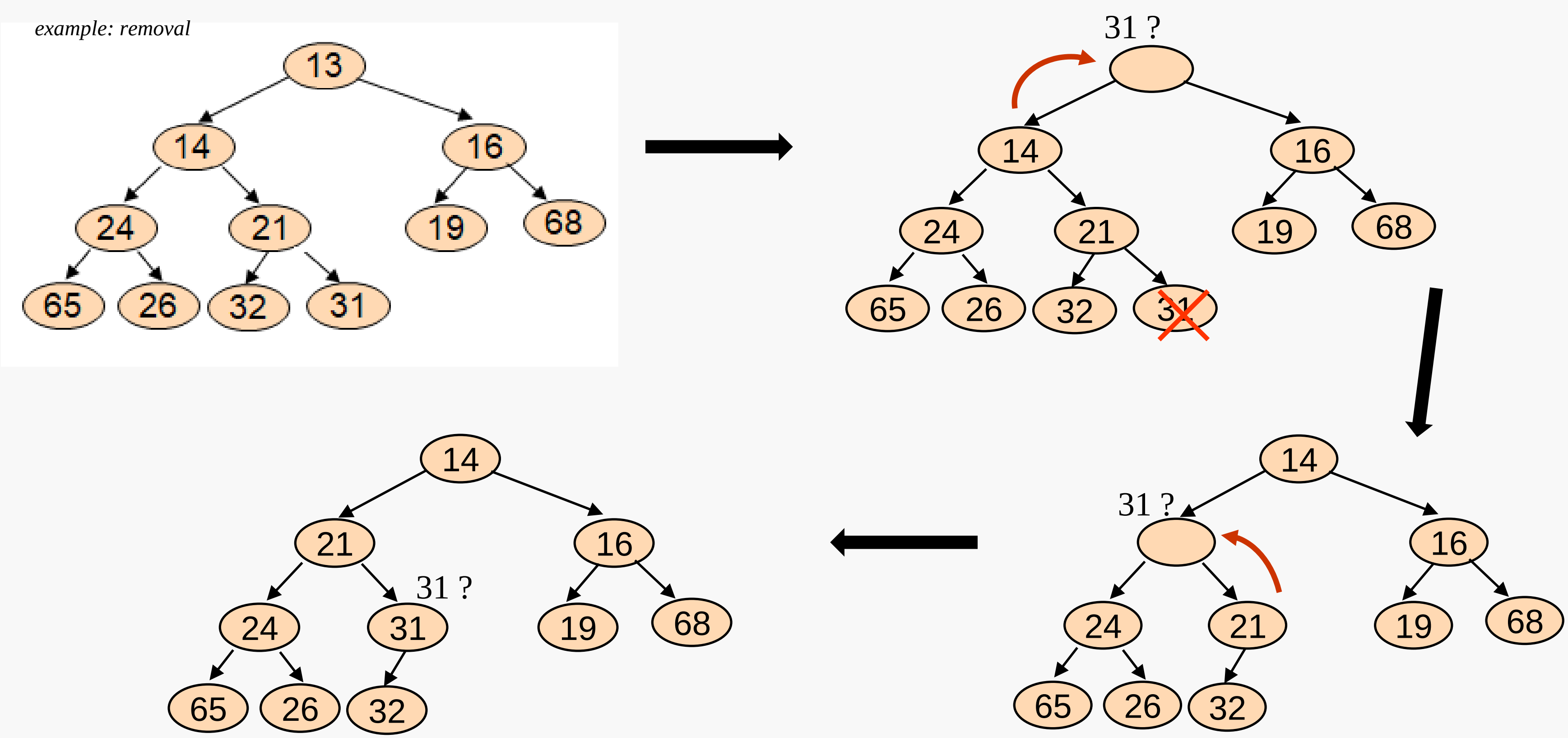




see animation in [VisuAlgo]()

Binary Heap

- Delete the element with highest priority: $O(\log n)$
  - element in the first position (root) is the highest priority
  - move the last element X to the first position
  - as long as the order is not respected: swap element X and smallest of its children



Heapsort: array sorting algorithm

- Algorithm
  - build a binary heap from the array : $O(n)$
  - do n operations removing the elements from the binary heap and store the elements successively in another vector: each operation has $O(\log n)$

$$T(n) = O(n \times \log n)$$

Problem/disadvantage:

- need to use another vector

Solution:

- use the same vector
- when an element is removed, the heap also frees a position; this position can be used to store the removed element.

Heapsort

Heapsort algorithm

```
template <class Comparable>
void heapsort(vector<Comparable>& a) {
    // build the heap
    for ( int i = a.size()/2; i >= 0; i--)
        percDown(a, i, a.size());
    //removals
    for ( int j = a.size() - 1; j > 0; j--){
        Comparable t = a[0];
        a[0] = a[j]; a[j] = t;
        percDown(a, 0, j);
    }
}
```

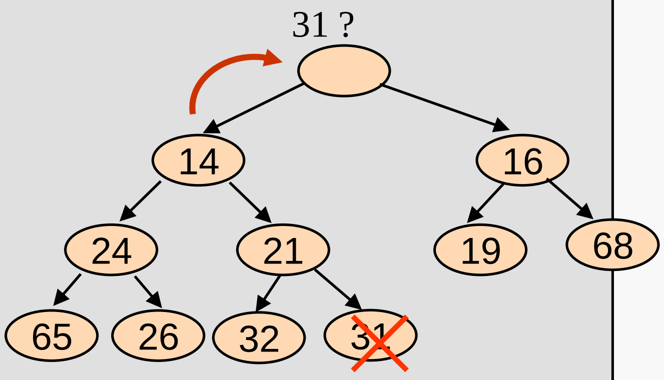
build the heap

n removals

Heapsort

percDown: element in position *i* moves down the tree until the heap property is satisfied

```
template <class Comparable>
void percDown(vector<Comparable>& a, int i, int n) {
    int child;
    Comparable tmp;
    for ( tmp = a[i]; (2*i + 1) < n; i = child ) {
        child = 2 * i + 1;
        if ( child != n-1 && a[child] < a[child+1] )
            child ++;
        if ( tmp < a[child] )
            a[i] = a[child];
        else
            break;
    }
    a[i] = tmp;
}
```



class *priority_queue* (STL)

class ***priority_queue*** in STL:

- implemented as a **max-heap**

- Some methods:
  - bool **empty**() const
  - int **size**() const
  - const T& **top**() const
  - void **push**(const T&)
  - void **pop**()

Priority Queue: example

- Resource allocation problem
  - Implement a program that allocates a set of tasks over several machines, in order to minimize the time it takes to execute all the tasks.

- LPT (“longest processing time first”) strategy
  - Tasks are allocated to machines in descending order of their processing time
  - Tasks are allocated to machines as they become free
  - To determine the first free machine, a priority queue is used, ordered according to the instant in which the machines are free.
  - To each machine removed from the queue, is allocated the next task, and the instant when the machine will be free again is calculated. The machine is then re-entered into the priority queue.

Priority Queue: example

```
struct Machine {
  int ID, free;
  bool operator < (const Machine& m)
const {
     return (free > m.free); }
};

struct Task {
  int ID, duration;
  bool operator < (const Task& t) const {
     return (duration < t.duration); }
};
```

```
int main() {
  vector<Task> tasks;
  read_tasks(tasks);
  int nm;
  cout << "Number of machines: "; cin >> nm;
  LPT(tasks, nm);
  return 1;
}
```

Priority Queue: example

```
void LPT(vector<Task>& a, int nm) {
  heapsort(a);
  priority_queue<Machine> h;
  Machine m1;
  for ( int i = 1; i <= nm; i++ ) {
       m1.free = 0; m1.ID = i;
     h.push(m1);
  }
  for ( int i = a.size()-1; i>=0; i-- ) {
     m1 = h.top();
     h.pop();
     cout  << a[i].ID << " in machine " << m1.ID << " from "
       << m1.free << " to " << (m1.free+a[i].duration) << endl;
     m1.free += a[i].duration;
     h.push(m1);
  }
}
```

ordered vector of tasks (increasing duration)

priority queue of machines