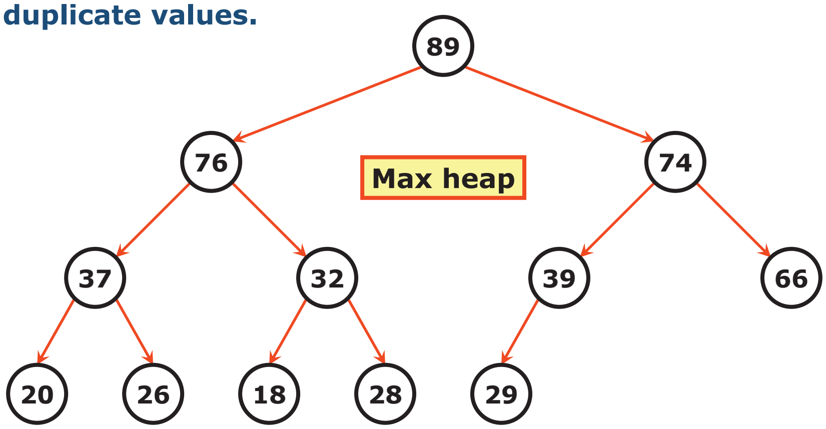
CPT-281 - Introduction to Data Structures with C++

Module 11

Heaps and Priority Queues

Dayu Wang

- **Heap**
  - → A complete binary tree is a max heap if:
    - The value in the root is the maximum value in the tree.
    - Every subtree is also a max heap.
  - → A complete binary tree is a min heap if:
    - The value in the root is the minimum value in the tree.
    - Every subtree is also a min heap.
  - → Heaps must be complete binary trees.
  - → Heaps can have duplicate values.



- **Inserting a Value into a Max Heap (Algorithm)**

  1) Insert the value in the "next position" across the bottom level of the complete tree.

     Preserve the completeness of the binary tree.

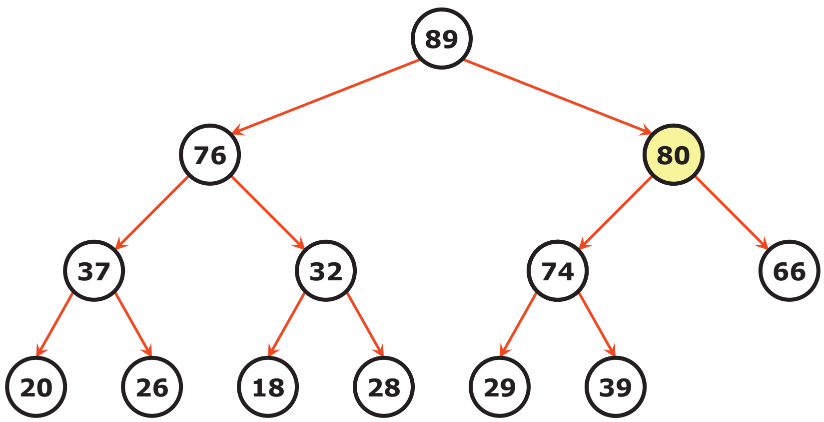
  2) Restore "heapness".

     while new_value is not the root and new_value > parent

       swap new_value with its parent

     endwhile

→ [Example] Insert **80** into the max heap.



- **Removing a Value from a Max Heap (Algorithm)**

→ You can only remove the root from a heap.

  1) Replace the root with bottom-level rightmost value (denoted as x).

     Preserve the completeness of the binary tree.

  2) Remove the bottom-level rightmost value.

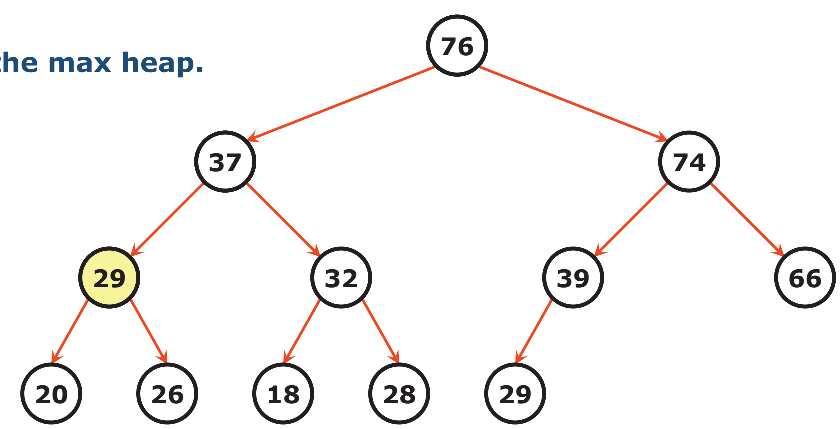
  3) Restore "heapness".

     while x has children and x is not greater than or equal to all children

       swap x with its larger child

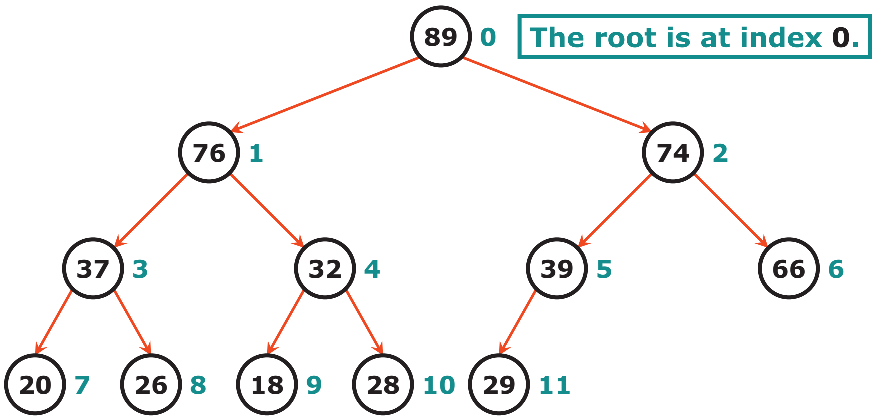
     endwhile

→ [Example] Remove the root from the max heap.



- **Implementation of Heaps**

→ **Since heaps are complete binary trees, so the most efficient way to implement them is using arrays.**



0	1	2	3	4	5	6	7	8	9	10	11
89	76	74	37	32	39	66	20	26	18	28	29

The array is essentially a level-order (row-major order) traversal.

- **Finding a Value's Children in the Array Presentation of Heaps**

0	1	2	3	4	5	6	7	8	9	10	11
89	76	74	37	32	39	66	20	26	18	28	29

→ **Which values are the children of 32?**

From the tree presentation of the heap, we know that:

The left child of 32 (index 4) is 18 (index 9).

The right child of 32 (index 4) is 28 (index 10).

→ **If the index of the parent is p, then:**

Its left child is at index 2p + 1.

Its right child is at index 2p + 2.

→ **[Pitfall] 2p + 1 and/or 2p + 2 may be out of bounds!**

[Example] 39 (index 5)

Its left child is 29 (index 11).

It does not have right child, since index 12 is out of bound.

Before going to a child, you need to first check whether the child exists or not.

- **Finding a Value's Parent in the Array Presentation of Heaps**

0	1	2	3	4	5	6	7	8	9	10	11
89	76	74	37	32	39	66	20	26	18	28	29

→ **Which value is the parent of 20?**

From the tree presentation of the heap, we know that the parent of 20 (index 7) is 37 (index 3).

→ **Which value is the parent of 26?**

From the tree presentation of the heap, we know that the parent of 26 (index 8) is also 37 (index 3).

→ **If the index of the child is c, then:**

Its parent is at index (c − 1) / 2.

→ **[Pitfall] The root does not have a parent!**

(0 − 1) / 2 == 0 in C++.

You may have a risk of infinite loop.

Before going to the parent, you need to check whether you have already reached the root.

- **Priority Queue**

→ **How to use heaps in C++?**

You need to use the build-in `priority_queue` **class in C++.**

`priority_queue` **is defined in the** `<queue>` **library.**

```
priority_queue<int> pq;
```

→ **Priority queue push/pop elements in priority order.**

→ **In C++, the priority queue is a max heap by default.**

→ **You need to use** `greater<int>` **if you need a min heap.**

```
priority_queue<int> pq_1;  // Max heap
priority_queue<int, vector<int>, greater<int>> pq_2;  // Min heap
```

Function	Behavior
`size_t size() const;`	Returns the size of the priority queue.
`bool empty() const;`	Returns `true` if priority queue is empty; `false` otherwise.
`T& top();`	Returns the value with highest priority (lvalue).
`const T& top() const;`	Returns the value with highest priority (rvalue).
`void pop();`	Deletes the value with highest priority.
`void push(const T&);`	Inserts a new value to the priority queue.

- **Performance of Heaps**

→ The insert algorithm traces a path from a leaf node to the root.

→ The remove algorithm traces a path from the root to a leaf node.

→ They both require h steps (h is the height of the tree).

→ Are heaps balanced binary trees?

Heaps are complete binary trees, which are balanced binary trees.

→ Therefore, both insert and remove are $O(\log n)$.

- **Heapsort**

→ Heapsort uses the properties of heaps to sort an array.

In this class, we will use array of integers as example.

→ Heapsort includes 3 algorithms:

- The Max-Heapify algorithm
- The Build-Max-Heap algorithm
- The Sort algorithm

In addition to the algorithms themselves, it is also important to understand and remember the pre-conditions to apply those algorithms.

- **Max Heapify**

→ **Condition to use the algorithm**

The entire array is a max heap everywhere, except one index, root.

→ **The max_heapify() function**

```cpp
/** Moves the root downward to form a max heap.
    @param vec: vector to sort
    @param root: index of the root
    @param size: size of the vector
*/
template<class T>
void Heapsort<T>::max_heapify(vector<T>& vec, size_t root, size_t size) {
    size_t left = root * 2 + 1, right = root * 2 + 2, max = root;
    if (left < size && vec.at(max) < vec.at(left)) { max = left; }
    if (right < size && vec.at(max) < vec.at(right)) { max = right; }
    if (max != root) {
        swap(vec.at(root), vec.at(max));
        max_heapify(vec, max, size);
    }
}
```


- **Build Max Heap**

→ **The algorithm converts an array to a max heap.**

→ **The build_max_heap() function**

```cpp
/** Generates a max heap rooted at given index.
    @param vec: vector to sort
    @param root: index of the root
*/
template<class T>
void Heapsort<T>::build_max_heap(vector<T>& vec, size_t root) {
    size_t left = root * 2 + 1, right = root * 2 + 2;
    if (left < vec.size()) { build_max_heap(vec, left); }
    if (right < vec.size()) { build_max_heap(vec, right); }
    max_heapify(vec, root, vec.size());
}
```

- **Sort**

→ **First, call the `build_max_heap()` function to convert the array to a max heap.**

0	1	2	3	4	5	6
1	2	5	6	9	11	13

Max heap

→ **Treat the entire array as the max heap, swap the first and last values.**

Since in the max heap, the root (index 0) is already the largest value, so this step will place the largest value at the end of the array.

→ **Shrink the size of the array by 1.**

The last value (13) is fixed.

In the "new array" (shrunk), it is a max heap everywhere except for the root (9).

We can call the `max_heapify()` function to make it a max heap.

→ **Swap the first and last values in this array.**

The second largest value in the whole array is placed appropriately.

→ **Shrink the size of the array by 1 and repeat previous steps, until the size of the array is 1.**

→ **The `sort()` function**

```
/** Sorts a vector of given size.
    @param vec: vector to sort
    @param size: size of the vector
*/
template<class T>
void Heapsort<T>::sort(vector<T>& vec, size_t size) {
    build_max_heap(vec, 0);
    for (size_t i = size - 1; i > 0; i--) {
        swap(vec.at(0), vec.at(i));
        max_heapify(vec, 0, --size);
    }
}
```

→ **Wrapper function**

```
// Wrapper function
template<class T>
void Heapsort<T>::sort(vector<T>& vec) { sort(vec, vec.size()); }
```

- **Performance of Heapsort**

→ **Building a max heap**

$\log 1 + \log 2 + \log 3 + \cdots + \log n = \log(n!)$

Property of logarithm: $\log x + \log y = \log(xy)$

Sterling's formula: $\log(n!) = O(n \log n)$

Therefore, building a max heap is $O(n \log n)$.

→ **Shrinking a heap**

$\log n + \log(n-1) + \log(n-2) + \cdots + \log 1 = \log(n!) = O(n \log n)$

→ **Time complexity of heapsort:** $O(n \log n)$

→ **Heapsort is in-place sorting algorithm.**

→ **Heapsort is unstable.**