5SENG003W - Algorithms, Week 7

Dr. Klaus Draeger

RECAP

Last week...

- We talked about the balanced trees
  - Why?
    - Guaranteed $O(\log(n))$ complexity for search, insertion, deletion
    - Unbalanced trees only guarantee $O(n)$
  - How?
    - Additional data in nodes: height, balance factor
    - Rotation operations
    - Left / Right / Left-right / Right-left

Overview of today's lecture

- Lists, queues and priority queues
  - Interfaces built on top of underlying data structures
  - Defined by specialised **restricted** access
  - This affects suitability of data structures
- Heaps
  - A different kind of ordered tree
  - Used in priority queues and the Heap Sort algorithm

Stacks

A **stack** is a data structure containing a collection of items of the same data type that **can only be accessed at one end**, known as the **top** of the stack.

Items can be:

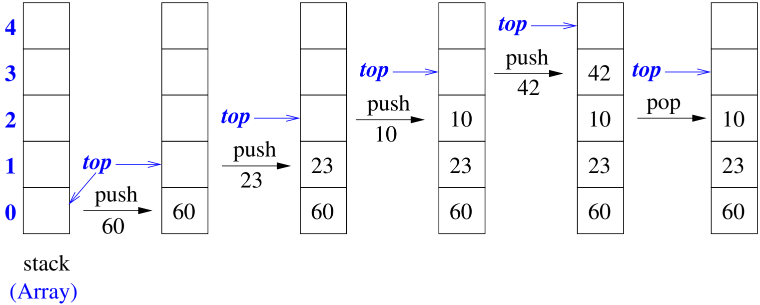
- "**pushed**" onto the stack – that is **added** onto the **top** of the stack,
- "**popped**" off the stack – that is **removed** from the **top** of the stack.

A **stack** is known as a **Last-In-First-Out (LIFO)** data structure. This is because the **last item added to the stack** by **push**, will be the **first item removed from it** by **pop**.
The "**top of a stack**" when it is implemented as:

Examples of Stacks

A **stack** produced by:

```
push(60) ;
push(23) ;
push(10) ;
push(42) ;
pop() ;
```



Implemented using an array, with **top** as the highest **unused index.**

Stacks

Uses of stacks include:

- Anything where you may need to "come back to" previous points (which are stored on the stack)
- Implementing **recursion** or generally nested **function calls**
  - When a function is called, a **call frame** is pushed on a stack
  - The frame underneath is for the calling function which is suspended, waiting for the result
  - When the call finishes, its frame is popped off the stack; the result is returned to the calling function which then resumes
- **Backtracking** searches
  - Trying to get to a target from a starting location
  - In the current location **L**:
    - If there is an unexplored neighbour **N**, push L onto the stack and continue from N
    - Otherwise we are stuck: pop previous location off the stack to check other neighbours

Example: Call Stack

```
int factorial(int n){
    if(n > 1)
        return n * factorial(n-1);
    return 1;
}
```

Factorial(3)

if(3 > 1)
    return 3 * factorial(2);
return 1;

Example: Call Stack

```
int factorial(int n){
    if(n > 1)
        return n * factorial(n-1);
    return 1;
}
```

```
Factorial(2)
---
if(2 > 1)
    return 2 * factorial(1);
return 1;
```

```
Factorial(3)
---
if(3 > 1)
    return 3 * factorial(2);
return 1;
```

Example: Call Stack

```
int factorial(int n){
    if(n > 1)
        return n * factorial(n-1);
    return 1;
}
```

```
Factorial(1)
if(1 > 1)
    return 1 * factorial(0);
return 1;
```

```
Factorial(2)
if(2 > 1)
    return 2 * factorial(1);
return 1;
```

```
Factorial(3)
if(3 > 1)
    return 5 * factorial(4);
return 1;
```

Example: Call Stack

```c
int factorial(int n){
    if(n > 1)
        return n * factorial(n-1);
    return 1;
}
```

Factorial(2)
if(2 > 1)
return 2 * 1;
return 1;

Factorial(3)
if(3 > 1)
return 3 * factorial(2);
return 1;

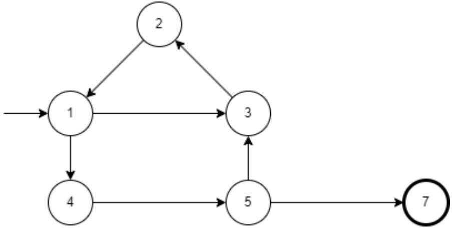
Example: Call Stack

```
int factorial(int n){
    if(n > 1)
        return n * factorial(n-1);
    return 1;
}
```

Factorial(3)
if(3 > 1)
return 3 * 2;
return 1;

Example: Backtracking

Trying to get from **1** to **7** in this graph:



could go like this:

- Try **1**'s neighbour **3**; stack is now **[1]**.
- Try **3**'s neighbour **2**; stack is now **[1,3]**.
- **2** is a dead end. Pop **3** off the stack which is now **[1]**.
- **3** is a dead end. Pop **1** off the stack which is now empty.
- Try **1**'s neighbour **4**; stack is now **[1]**.
- And so on.

Stacks

Stacks are a **sequential** data structure and can be implemented using indexed or linked data structures, i.e.

- an **array** (ArrayList, vector, ...) with
  - a current (possibly fixed, for a plain array) **capacity**
  - a **size** which is the number of currently stored values
  - The size is also the **top** index where the next pushed element will go
- a **list** where
  - the **top** is the **first node**.
  - For this purpose a **singly-linked** list is enough.

Example: List-based Stack, part 1

```
import java.util.EmptyStackException;

public class ListStack{
    class ListNode{
        // Contents of this node
        public int data;
        // The node below this one
        public Listnode next;
        public ListNode(int d, ListNode n){
            data = d;
            next = n;
        }
    }

    // Top of the stack; null if stack is empty.
    private ListNode topNode;

    public ListStack(){
        topNode = null;
    }

    // ...
```

Example: List-based Stack, part 2

```
    // Get the top element
    // Throw an exception if the stack is empty.
    public int top(){
        if(topNode == null) // stack is empty
            throw new EmptyStackException();
        return topNode.data;
    }

    // Remove the top node, the one below is the new top.
    // Throw an exception if the stack is empty.
    public void pop(){
        if(topNode == null) // stack is empty
            throw new EmptyStackException();
        topNode = topNode.next;
    }

    // Add a new node which becomes the top
    public void push(int data){
        topNode = new ListNode(data, topNode);
    }
}
```

Queues

A **queue** is a data structure containing a collection of values of the same data type which can be **accessed at both ends**:

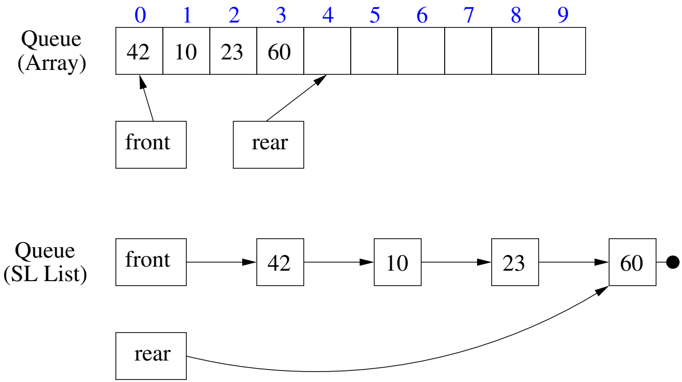
- data items are **queued** (or inserted) at the **back**,
- data items are **dequeued** (or removed) from the **front.**

The concept of a queue data structure in computing mirrors that in real-life.
A **queue** is a **First-In-First-Out (FIFO)** data structure: items are removed in the same order as they are inserted.

Queues Example

A **queue** produced by:

```
queue(42) ;
queue(10) ;
queue(23) ;
queue(60) ;
```



Example: Array-based Queue

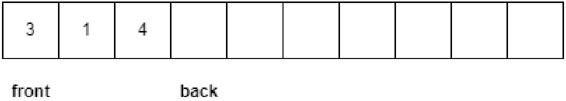
Array-based queue, version 1:

```
import java.util.RuntimeException;

public class ArrayQueue{
    public static void capacity = 10;
    private int[] entries;
    int front = 0, back = 0;

    public ArrayQueue(){
        entries = new int[capacity];
    }

    public void queue(int n){
        if(back == capacity)  // Out of spaces
            throw new RuntimeException();
        entries[back++] = n;
    }

    public int dequeue(){
        if(front == back)  // Empty queue
            throw new RuntimeException();
        return entries[front++];
    }
```

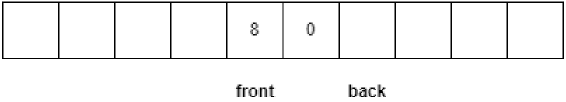
Example: Array-based Queue

This first version has a problem. For example:

- After the first few insertions:



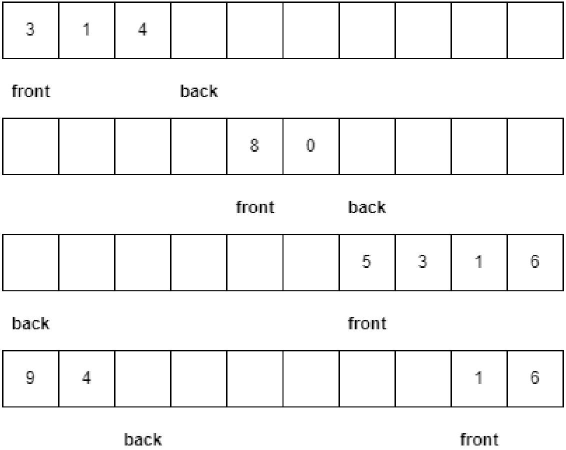
- After some more insertions and removals:



And we cannot enqueue any more values!
Using ArrayList/vector fixes this, but we still get an ever-growing set of "used up" indices.

Cyclic Queues

By allowing **`front`** and **`back`** to wrap around, we can **re-use** array entries:



. . . as long as we don't run out of space

Example: Array-based Queue

Array-based queue, version 2:

```
public class ArrayQueue{
    public static void capacity = 10;
    private int[] entries;
    int front = 0, back = 0;

    public ArrayQueue(){
        entries = new int[capacity];
    }

    public void queue(int n){
        if((back+1) % capacity == front)
            throw new RuntimeException();
        entries[back] = n;
        back = (back+1) % capacity;
    }

    public int dequeue(){
        if(front == back)  // Empty queue
            throw new RuntimeException();
        return entries[front];
        front = (front+1) % capacity
    }
```

Example: Array-based Queue

Still to do:

- Enable the queue to grow instead of throwing an exception when out of space
  - Replace the array with a bigger one (e.g. doubling the size)
  - Not a straightforward copy, need to shift appropiately
- Alternatively: Use a **list** based version
  - Tutorial!

Priority Queues

A **priority queue** stores a collection of **(key, data)** values in ascending (or descending) **key** order.

Its **main purpose** is to allow the fast **extraction** of the **"highest"** priority item, i.e. the one with the **minimum key** (we can use the reverse ordering if we want the maximum key instead)

The other main operation is the **insertion** of **(key, data)** values into its appropriate position according to its key value.

The underlying data struxture is usually a **heap**.

Heaps – Implementing Priority Queues

A priority queue is usually implemented using a **heap** which holds **(key, data)** values in either:

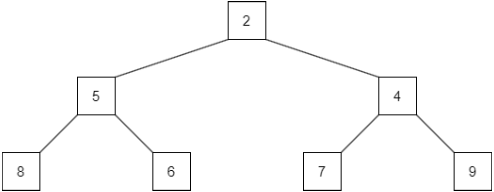
- **ascending** order (**smallest key** at the front, a **min-heap**).
- **descending** order (**largest key** at the front, a **max-heap**).

We will just look at min-heaps; max-heaps work essentially the same way.
The “**logical**” view of a heap data structure is that of a **binary tree**

- Not a binary search tree!
- The requirements make another structure more suitable
- There are two defining properties of a heap.

Defining Heaps

The **first property** of a (min-)heap is that each node's key is **smaller** than those of its children



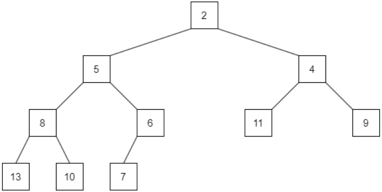
This means that

- The **minimum** value is always in the **root** node.
- We only care about the ordering **along each branch**, not between branches

Defining Heaps

The **second property** is that it has a special structure:

- All levels except the last one are completely filled
- The last level is filled from the left
- So if you read the nodes level by level, left to right, there would be no gaps

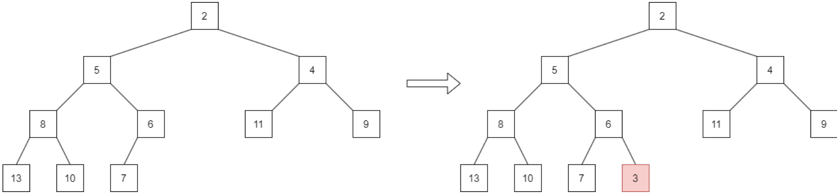


- This is possible because the ordering is more flexible (keys within a level can occur in any order)
- It will be important when implementing the heap.

Heap Operations: Insertion

When inserting a new value we need to preserve the defining properties

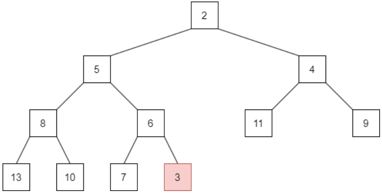
- For the **structure** property, we can simply place the value in the necessary position
  - The next position in the final level if not yet complete
  - The first position in a new level otherwise



Heap Operations: Insertion

When inserting a new value we need to preserve the defining properties

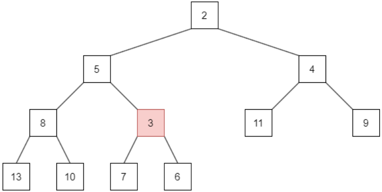
- Next we need to restore the ordering
  - Only need to handle the branch containing the new leaf
  - Essentially perform one iteration of Bubble Sort on this branch
  - This is called **Sifting Up**.



Heap Operations: Insertion

When inserting a new value we need to preserve the defining properties

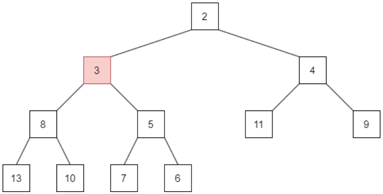
- Next we need to restore the ordering
  - Only need to handle the branch containing the new leaf
  - Essentially perform one iteration of Bubble Sort on this branch
  - This is called **Sifting Up**.



Heap Operations: Insertion

When inserting a new value we need to preserve the defining properties

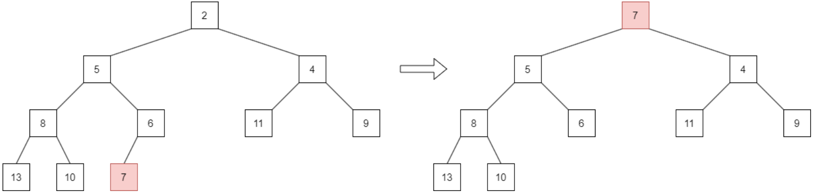
- Next we need to restore the ordering
  - Only need to handle the branch containing the new leaf
  - Essentially perform one iteration of Bubble Sort on this branch
  - This is called **Sifting Up**.



Heap Operations: Minimum Extraction

When extracting the minimum we again need to preserve the defining properties

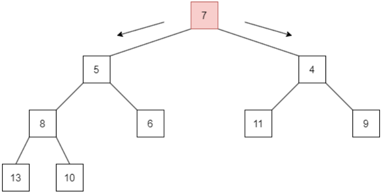
- For the **structure** property, we need to replace the extracted value
- The obvious choice is the rightmost value on the last level



Heap Operations: Minimum Extraction

When extracting the minimum we again need to preserve the defining properties

- Next we need to restore the ordering
- This time we want to sift the value **down**
- But we have two options: **left** or **right**

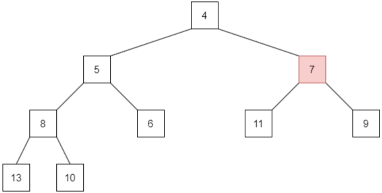


- Need to choose the **smaller** child, otherwise we would break the ordering again

Heap Operations: Minimum Extraction

When extracting the minimum we again need to preserve the defining properties

- Next we need to restore the ordering
- This time we want to sift the value **down**
- But we have two options: **left** or **right**



- Need to choose the **smaller** child, otherwise we would break the ordering again

Heap Operations: Minimum Replacement

There is a third operation which is sometimes useful:
When extracting the minimum, we may at the same time have a new value to be inserted. In this case,

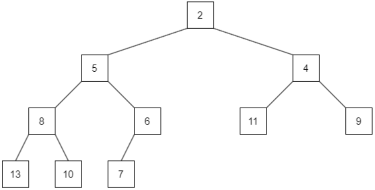
- Replace the minimum in the root by the new value instead
- Sift down as before

We will see one example where this occurs later.

Implementing Heaps

The strict level structure allows us to implement heaps as an **indexed** data structure backed by an array:

- The values are stored "level by level":
  - Index 0 contains the root
  - Indices 1 and 2 contain its children
  - indices 3-6 contain its grandchildren, etc

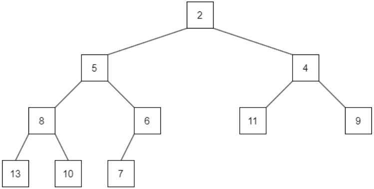


Index	0	1	2	3	4	5	6	7	8	9	
Key	2	5	4	8	6	11	9	13	10	7	...

- This means we do not need to deal with actual nodes
- But we do need to figure out details of the sifting operations in terms of array indices

Implementing Heaps

Looking at the layout of a heap more closely:



Index	0	1	2	3	4	5	6	7	8	9	
Key	2	5	4	8	6	11	9	13	10	7	...

We can see that for a node at position k,

- Its children are always at positions $2 * k + 1$ and $2 * k + 2$
- Its parent is always at position $(k - 1)/2$
- For example, the node with key 5 is at position 1 and
  - Its children (keys 8 and 6) are at positions $3 = 2 * 1 + 1$ and $4 = 2 * 1 + 2$
  - Its parent (key 2) is at position $0 = (1 - 1)/2$

Implementing Heaps: Example

The insertion in a heap (using just ints as values for simplicity) could look like this:

```
public class MinHeap{
    int[] items;

    public static final int INITIAL_CAPACITY = 1000;
    int size;  // Number of values currently in the heap

    public void insert(int newItem){
        // Determine position for insertion:
        // begin at index==size, then sift up
        int index = size++;
        while(index > 0){
            int parent = (index - 1) / 2;
            if(items[parent] <= newItem){ // place here
                items[index] = newItem;
                return;
            }
            items[index] = items[parent];
            index = parent;
        }
    }
```

Heap Sort

One use of a heap is the **Heap Sort** algorithm.
A simple version would be:

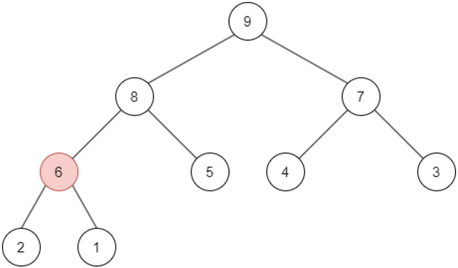
- Insert the data from the given array into a heap one at a time
- Keep extracting the minimum element from the heap until it is empty(storing them in order)
- This will give you the data in ascending order

Both insertion and minimum extraction work in time $O(\log(n))$, so this is in $O(n\log(n))$ overall
We cannot go below $O(n\log(n))$, but there is an improved version which converts the array into a heap in time $O(n)$.

Heap Sort: the Heapify Operation

The **heapify** operation transforms an array into a heap from the bottom up:

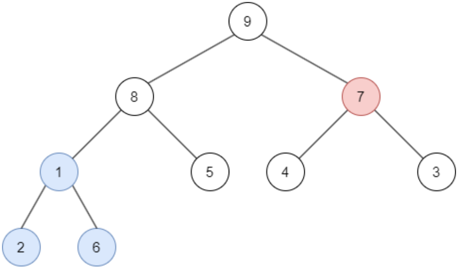
- Start at the last **non-leaf**, i.e. at the parent index $k := (size/2)$
- Decrease k after each iteration until it reaches -1
- In each iteration, **sift down** the element at position k; after this the subtree below position k is a heap.



Heap Sort: the Heapify Operation

The **heapify** operation transforms an array into a heap from the bottom up:

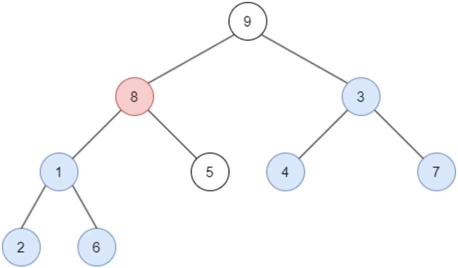
- Start at the last **non-leaf**, i.e. at the parent index $k := (size/2)$
- Decrease k after each iteration until it reaches -1
- In each iteration, **sift down** the element at position k; after this the subtree below position k is a heap.



Heap Sort: the Heapify Operation

The **heapify** operation transforms an array into a heap from the bottom up:

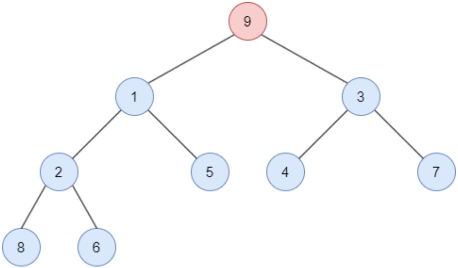
- Start at the last **non-leaf**, i.e. at the parent index $k := (size/2)$
- Decrease k after each iteration until it reaches -1
- In each iteration, **sift down** the element at position k; after this the subtree below position k is a heap.



Heap Sort: the Heapify Operation

The **heapify** operation transforms an array into a heap from the bottom up:

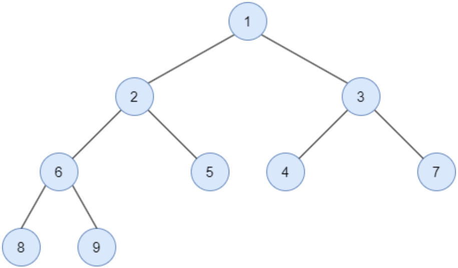
- Start at the last **non-leaf**, i.e. at the parent index $k := (size/2)$
- Decrease k after each iteration until it reaches -1
- In each iteration, **sift down** the element at position k; after this the subtree below position k is a heap.



Heap Sort: the Heapify Operation

The **heapify** operation transforms an array into a heap from the bottom up:

- Start at the last **non-leaf**, i.e. at the parent index $k := (size/2)$
- Decrease k after each iteration until it reaches -1
- In each iteration, **sift down** the element at position k; after this the subtree below position k is a heap.



Heap Sort: the Heapify Operation

Why is the heapify operation in $O(n)$? Sifting down still takes time in $O(\log(n))$, after all!
Consider a complete height 5 heap. It has 31 nodes:

- 16 leaves which do not get sifted down
- 8 nodes which may get sifted down once
- 4 nodes which may get sifted down twice
- 2 nodes which may get sifted down three times
- 1 node which may get sifted down four times

So the number of siftings is at most

$$\begin{array}{rl} & 1 * 8 + 2 * 4 + 3 * 2 + 4 * 1 \\ = & (8 + 4 + 2 + 1) + (4 + 2 + 1) + (2 + 1) + 1 \\ < & 16 + 8 + 4 + 2 \\ < & 31 \end{array}$$

i.e. less than the number of nodes.