Coventry University

Data Structures (Stacks, Queues, Arrays and Linked Lists)

Dr. Diana Hintea

Lecturer in Computer Science

diana.hintea@coventry.ac.uk

210CT Week 4

Learning Outcomes

1. Coursework and Recap
2. Stacks
3. Queues
4. Arrays
5. Linked Lists

Coursework and Recap

Coursework!!!!!

1. Harmonic Series(block 6)
2. Pseudocode for linear search and duplicate search (block 3) AND their corresponding Big-O (block 4)
3. Searching for an element within a certain range in an array (both sorted and unsorted) (block 5)
4. Node Delete (block 8)

- Deadline: 30th of October 2015 11.55 pm

Coursework and Recap

Tips

- if (task == PSEUDOCODE) then PSEUDOCODE
- If (task == CODE) then CODE
- Style, presentation
- Submission
- Plagiarism!!!

Coursework and Recap

Introduction

- Data structures are constructions for storing and organising data.
- An array is a common data structure.
- Elements stored in ordered contiguous sequence, accessed by index.
- Having the right structure to data can make a big difference.
- The heap from last week adds a new layer of structure that allows us to efficiently find the largest item in an array.
- Other structures include:
  - Stacks; Queues; Trees(Binary, Vantage-point, Red-black, A-B, etc.) Hash maps; Graphs.

Coursework and Recap

RECAP – the Heap

Making the Heap

- A heap is a binary tree in which each node is greater-than or equal to its children, should it have any. (Or the other way around, but we're using it this way for this example).
- A binary tree is a tree in which each node has 0, 1 or 2 children.

Coursework and Recap

How do we represent a tree in an array?

- Root is at position 0.
- Children at 1 and 2.
  - Children of 1 at 3 and 4.
  - Children of 2 and 5 and 6.
- General rule: children of i are at 2i+1 and 2i+2.
- For a given node at position i, its parent is at $\lfloor i/2 \rfloor$
  - The funny brackets mean "floor", or "rounded down".

Coursework and Recap

Pseudocode

```
HEAP-SORT(A)
heapify(A)
end ← length(A)-1
while end > 0
    #We know the largest is at A[0]...
    swap(A[end],A[0])
    #The end is in the right place, so move down 1
    end ← end - 1
    #Now fix the heap, since we moved the value out
    siftDown(A,0,end)
```

Coursework and Recap

Making the heap

```
HEAPIFY(A, count)
#Start from last parent...
start ← floor ((count - 2 ) / 2)

while start ≥ 0 do
    #Sift the value at start down
    #This picks up the item and puts it in the right place
    siftDown(A, start, count-1)
    #go to the next parent node
    start ← start - 1
```

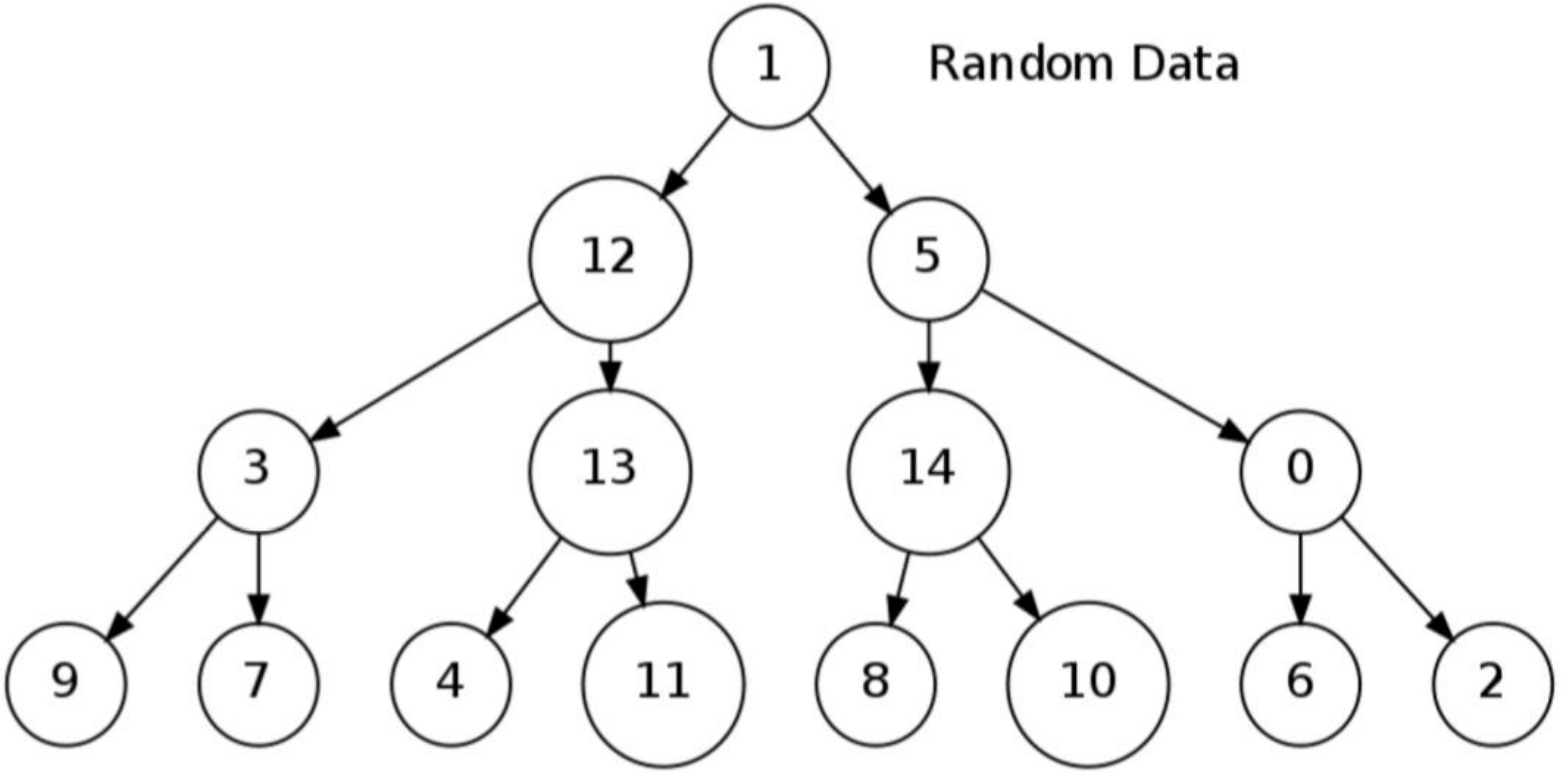
Coursework and Recap

Given the subtree, move the root down to the right place

```
SIFTDOWN(A, start, end)
root ← start
while root * 2 + 1 ≤ end #If there are children...
    swap ← root #Keep track of largest
    child ← root * 2 + 1  #Left child, is it larger?
    if A[swap] < A[child]
        swap ← child
    if child+1 ≤ end and A[swap] < A[child+1] #Right child larger?
        swap ← child + 1
    if swap ≠ root
        swap(A[root], A[swap])
        root ← swap
    else
        return #No changes, so must be done
```

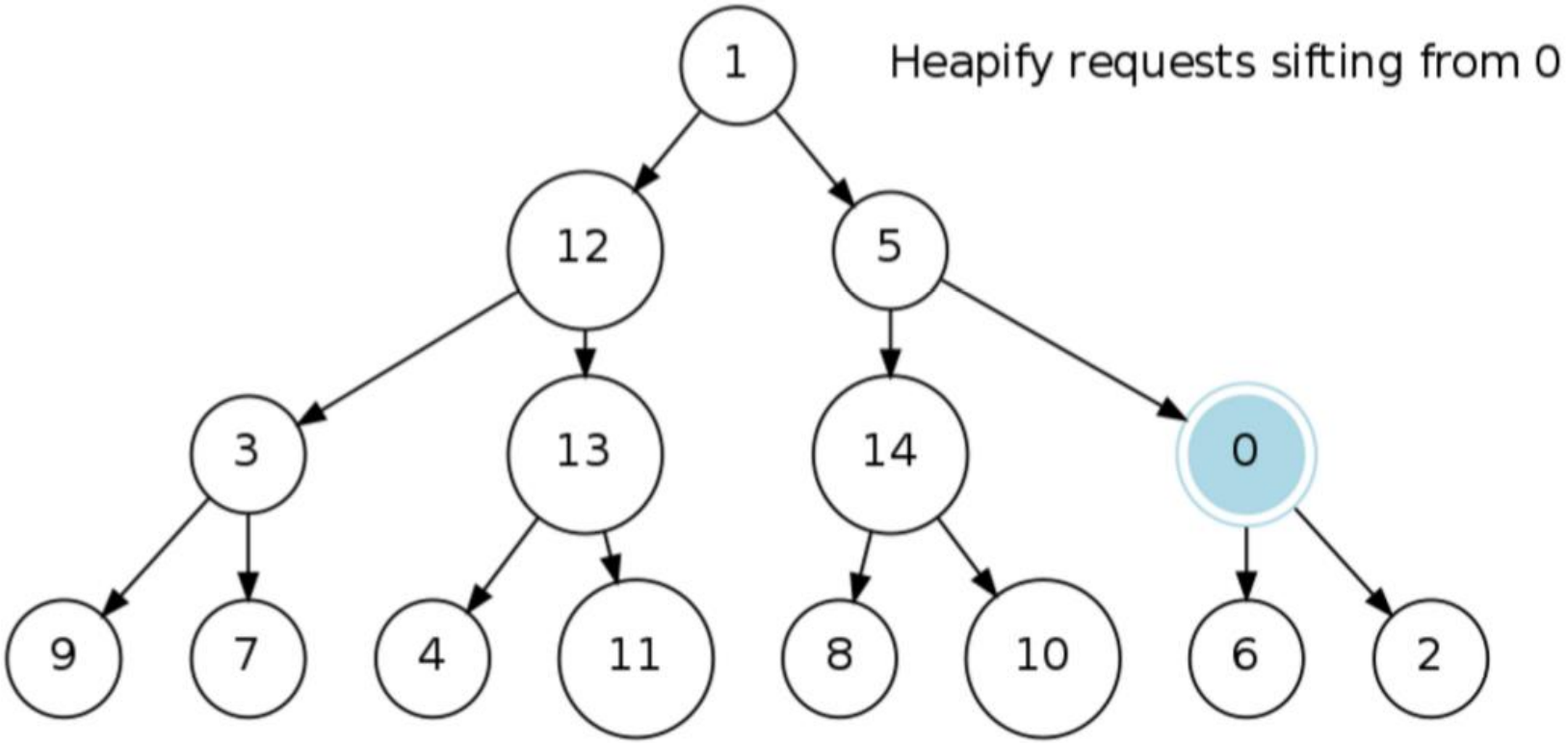
Coursework and Recap

Let's heapify



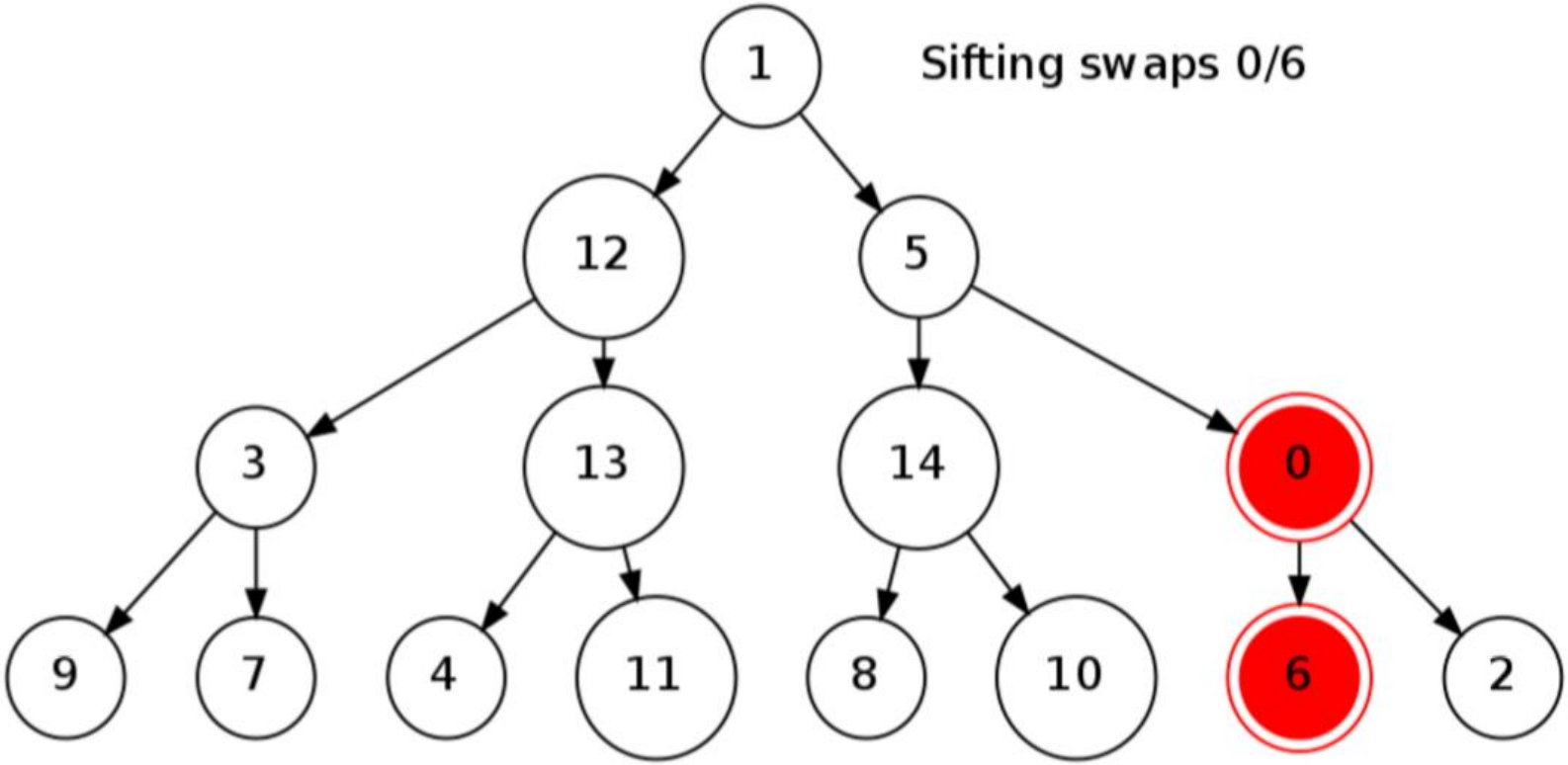
Coursework and Recap

Let's heapify



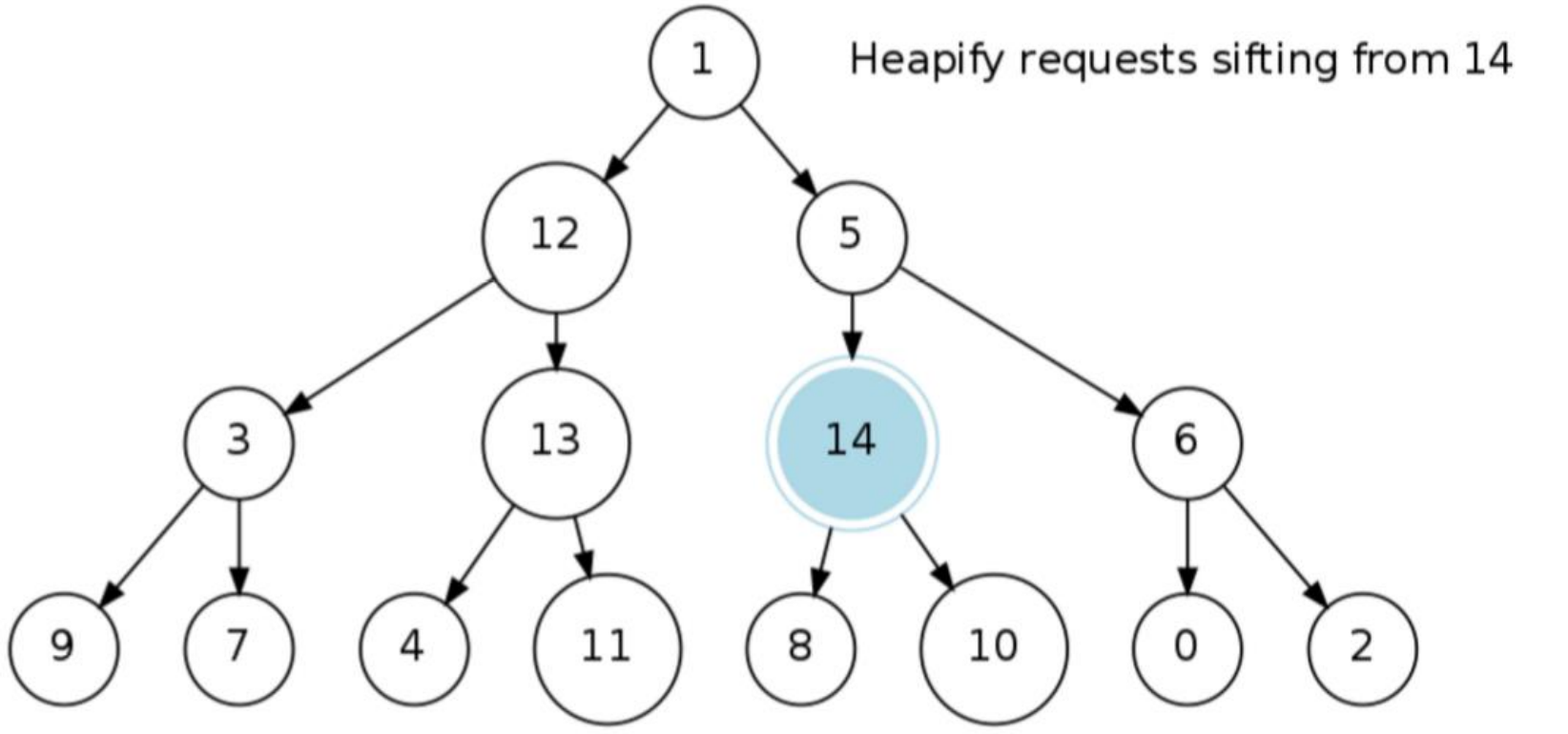
Coursework and Recap

Let's heapify



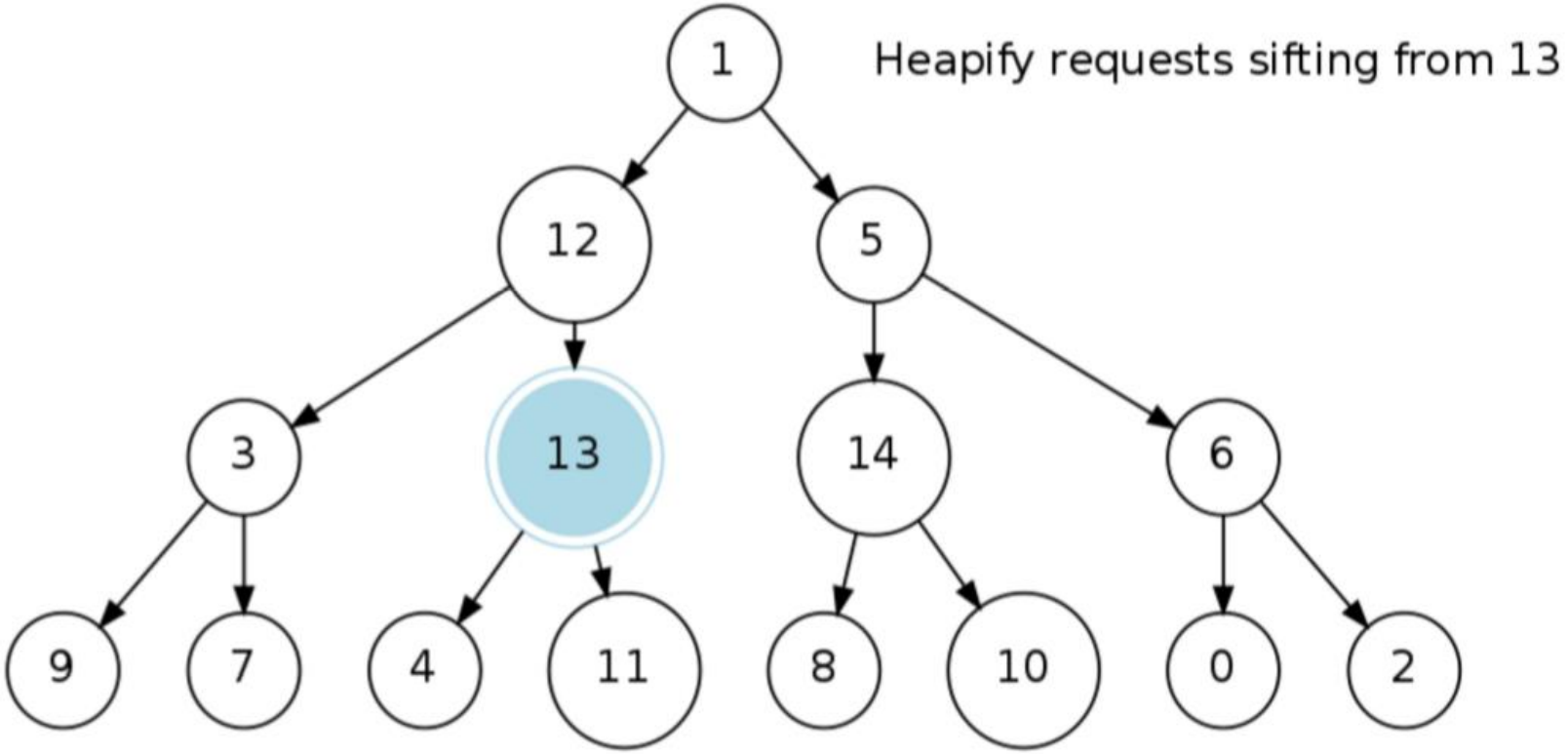
Coursework and Recap

Let's heapify



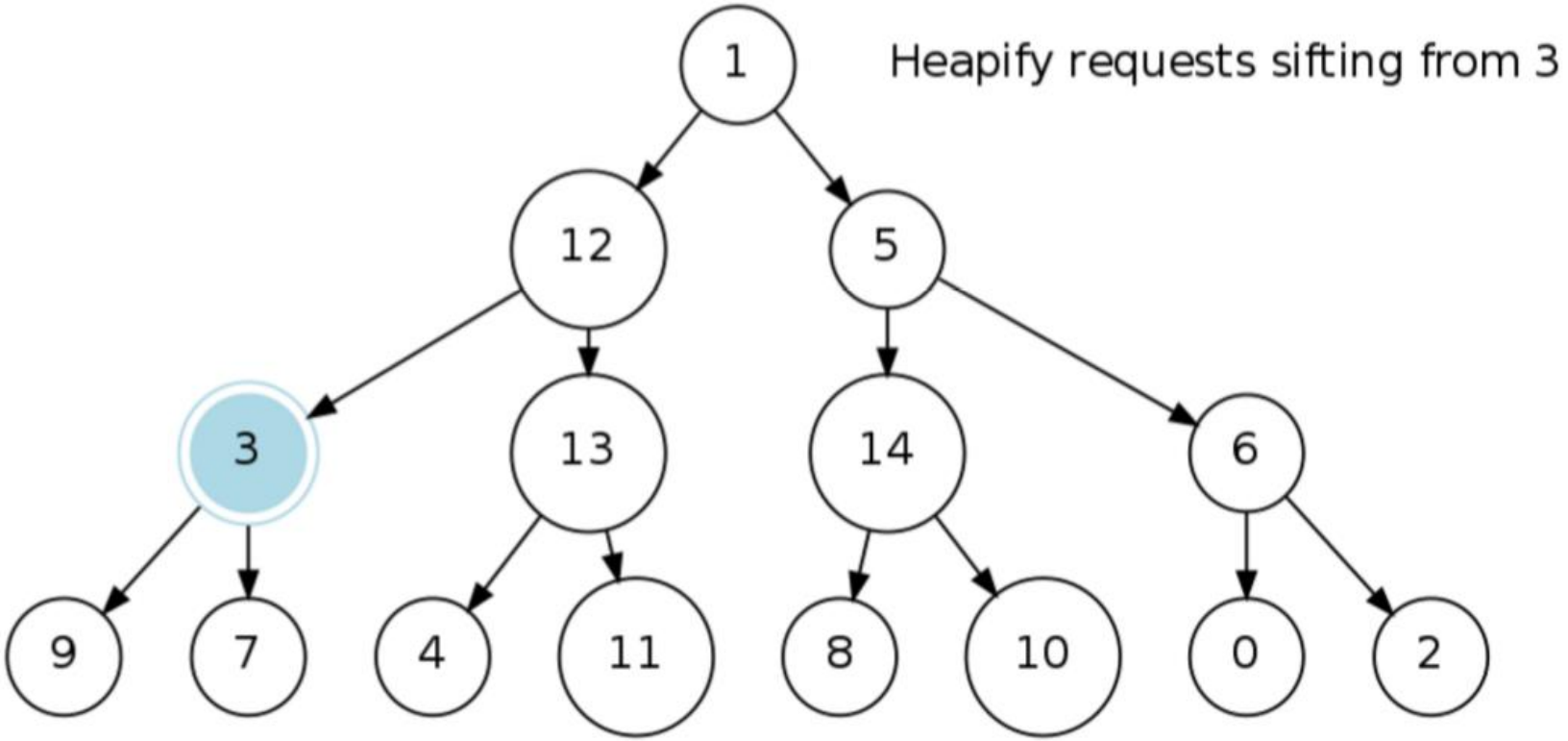
Coursework and Recap

Let's heapify



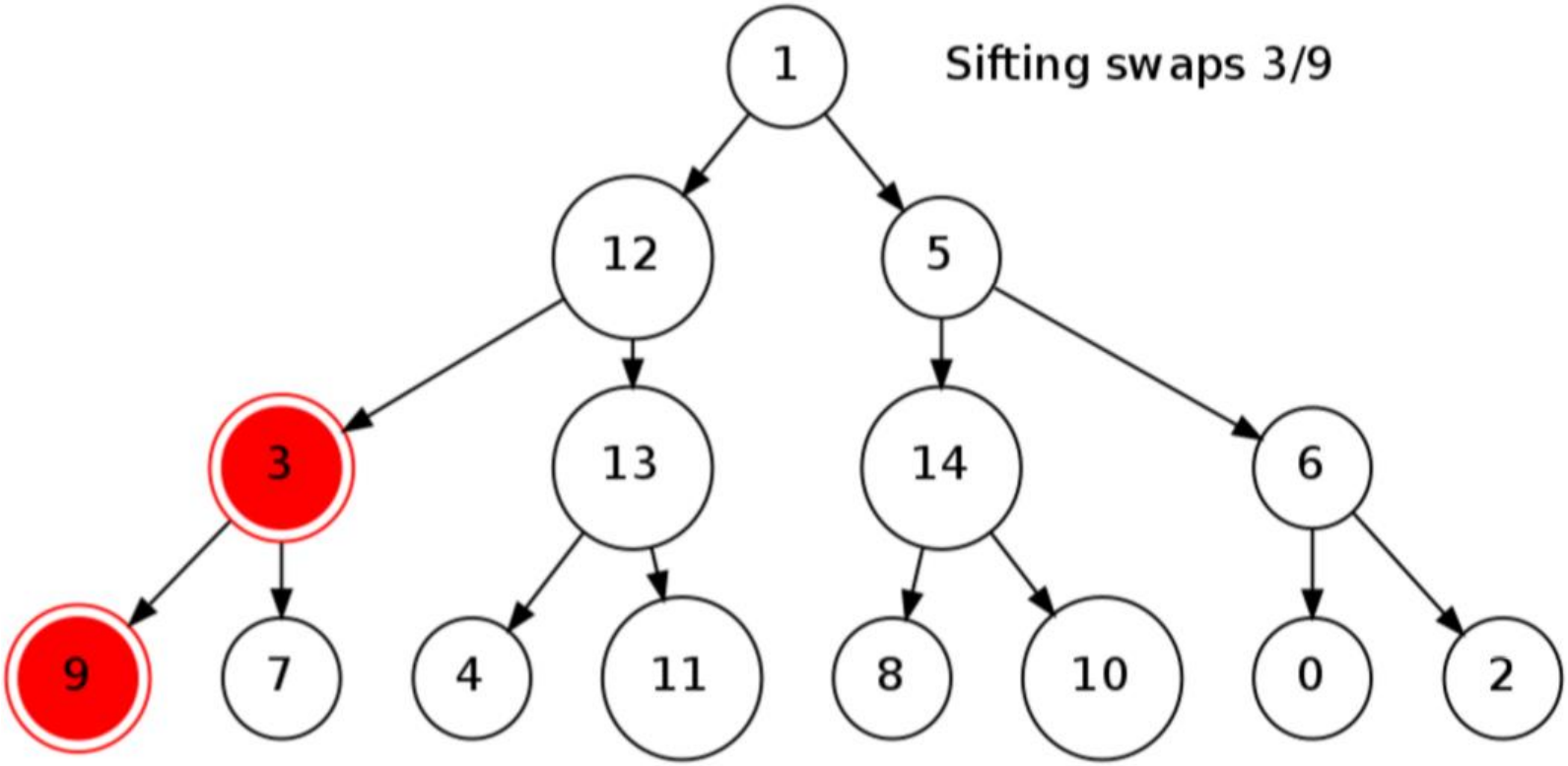
Coursework and Recap

Let's heapify



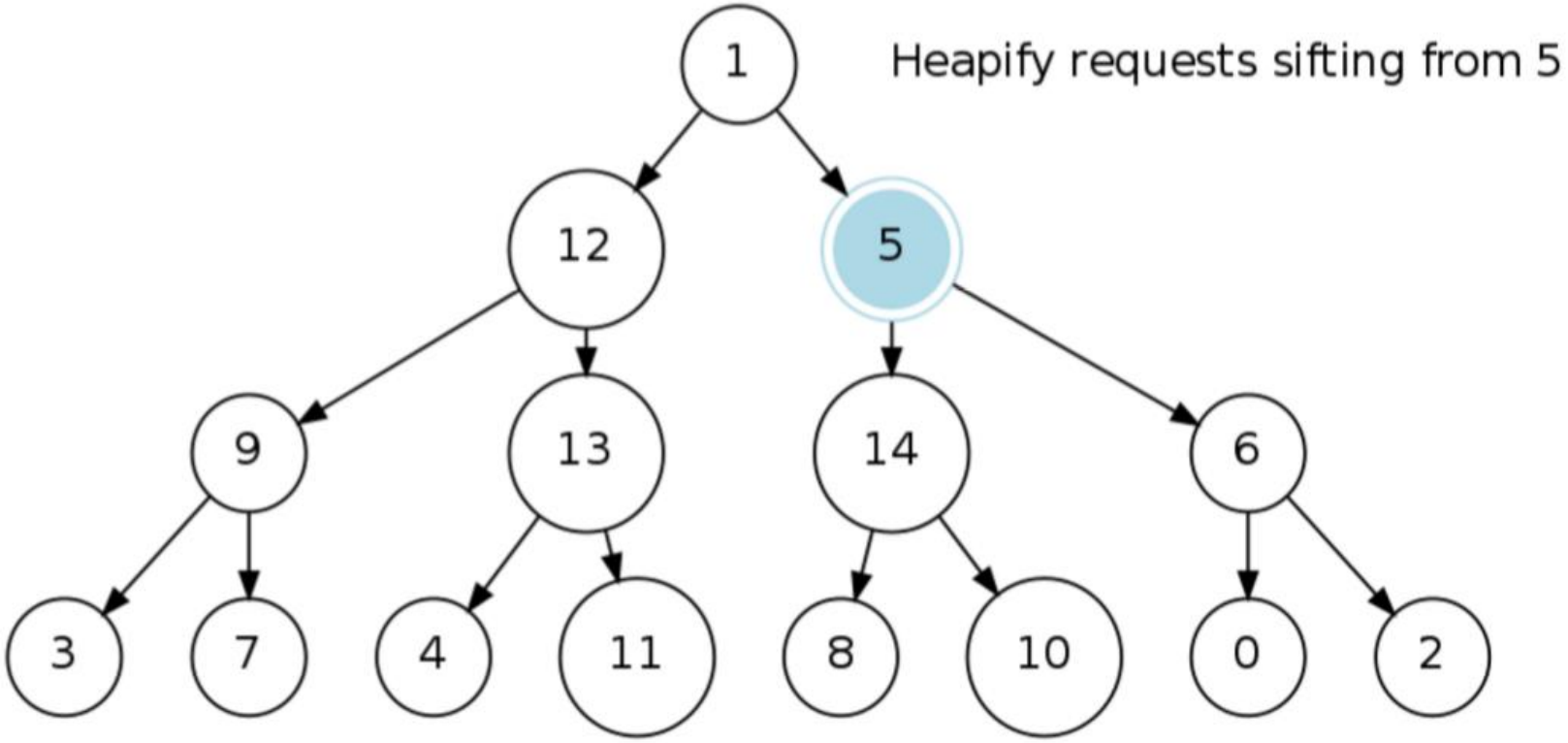
Coursework and Recap

Let's heapify



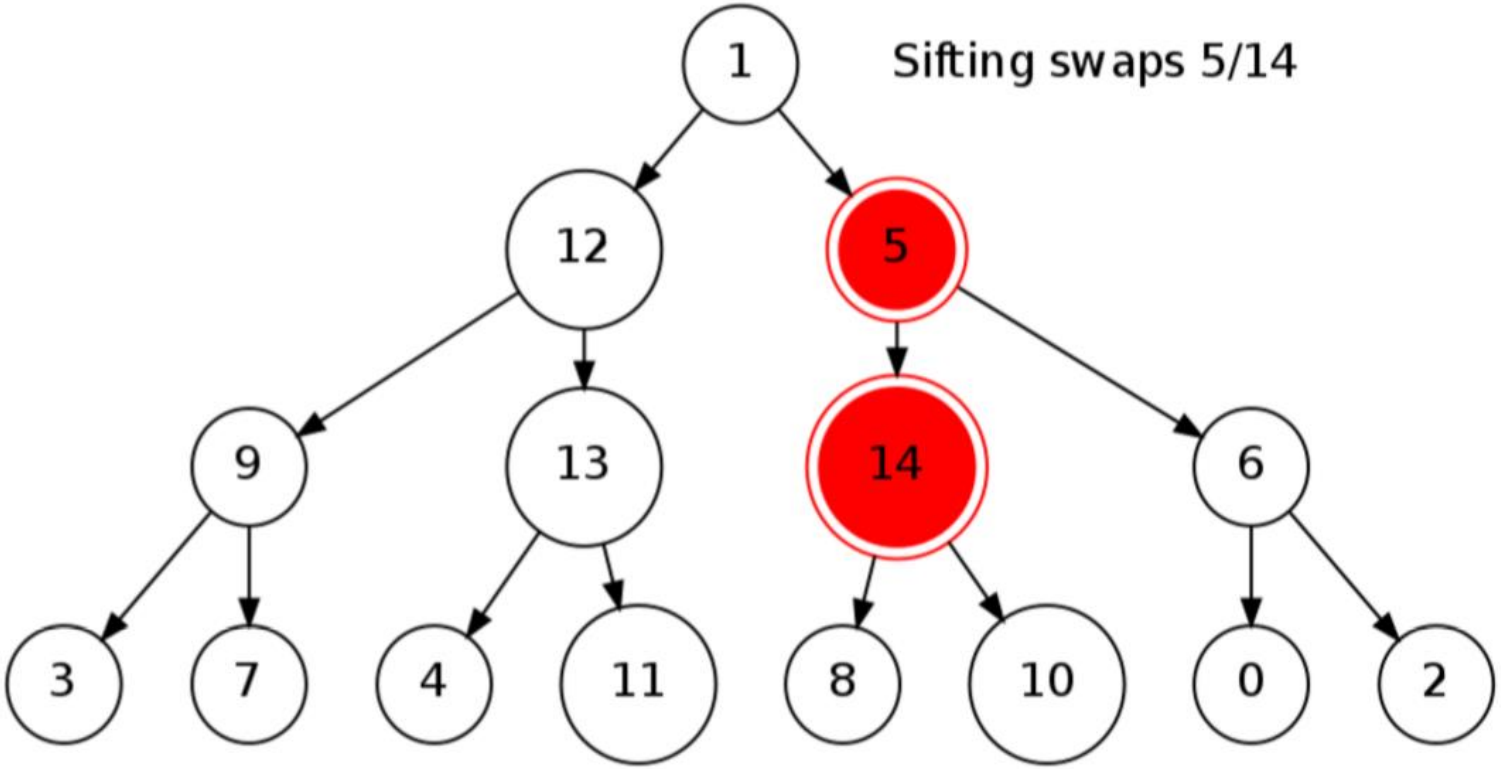
Coursework and Recap

Let's heapify



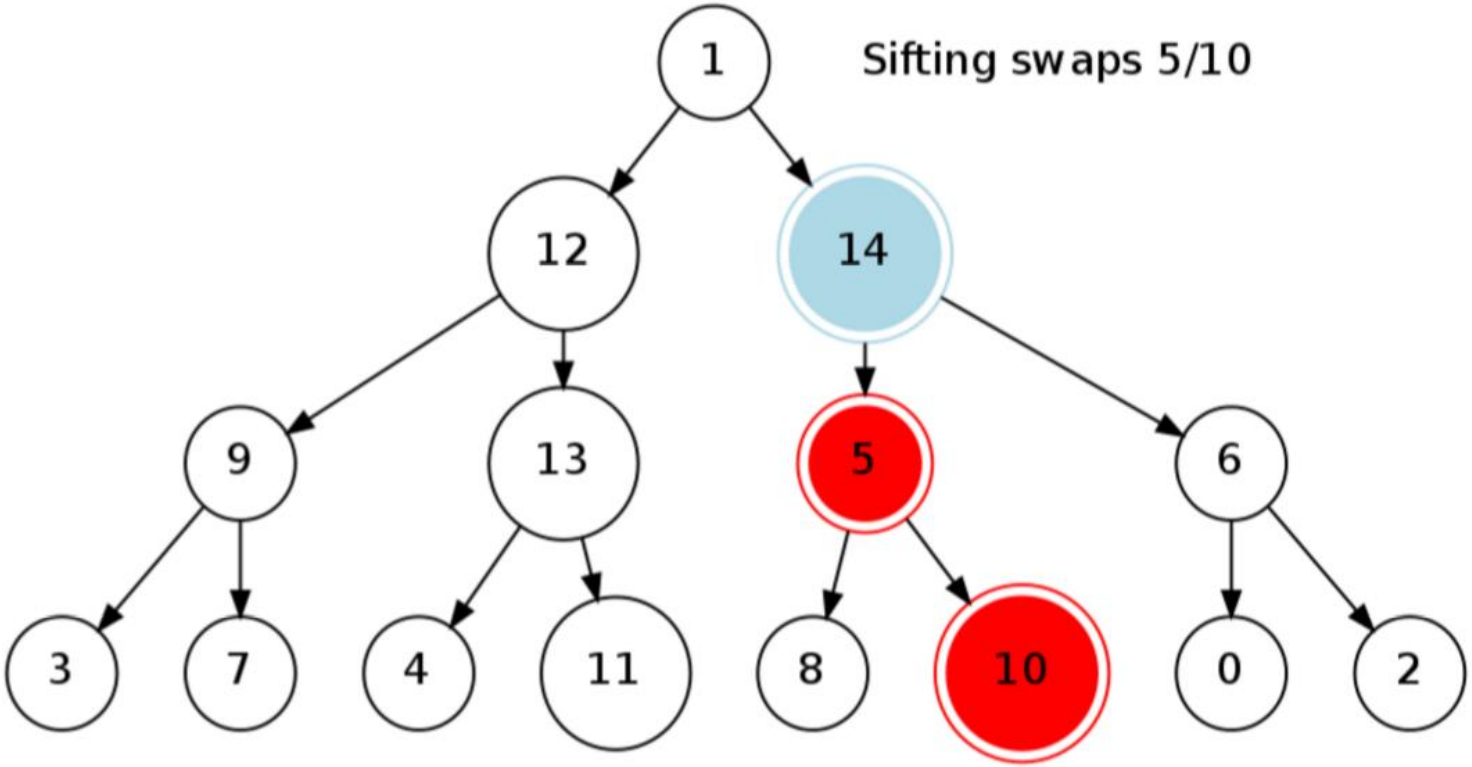
Coursework and Recap

Let's heapify



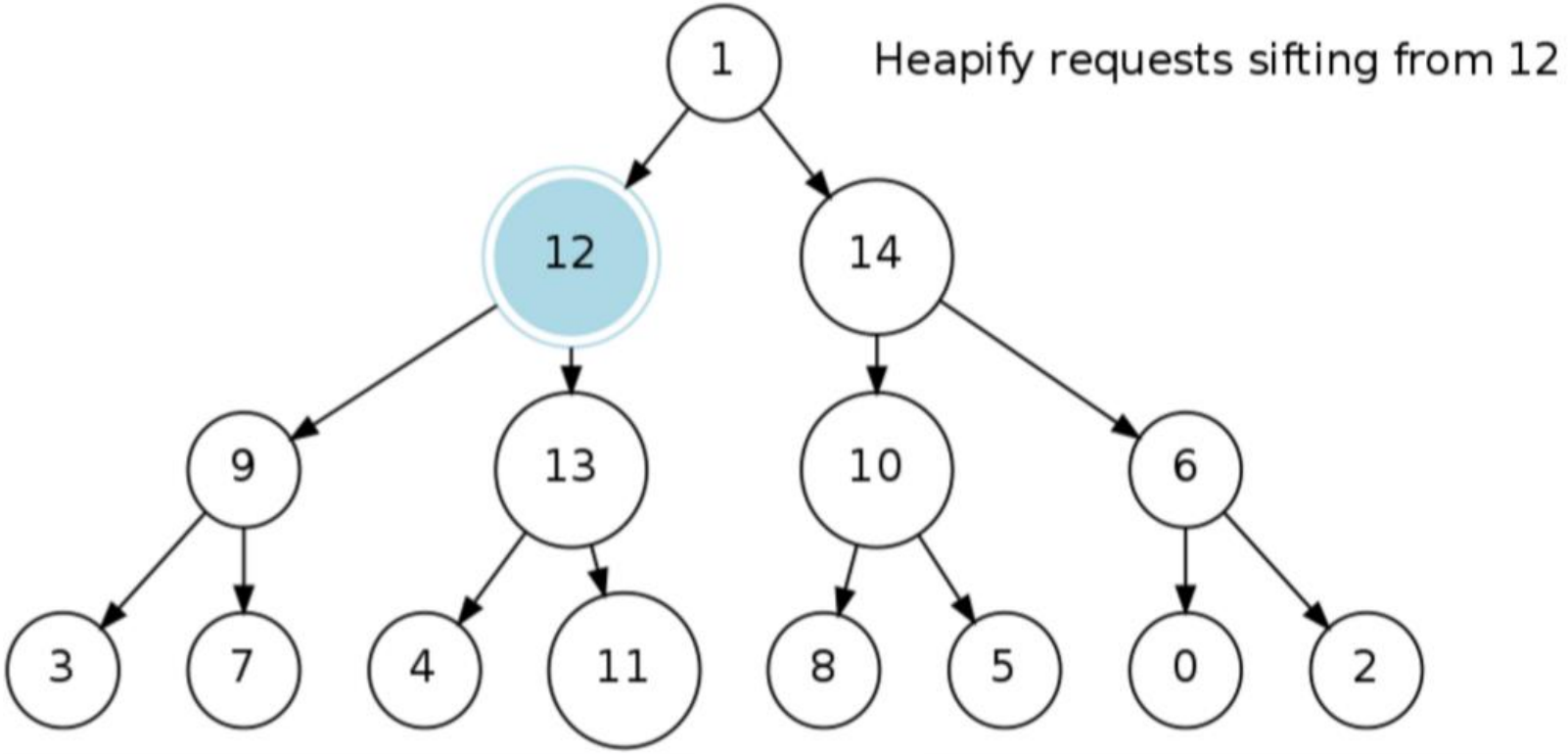
Coursework and Recap

Let's heapify



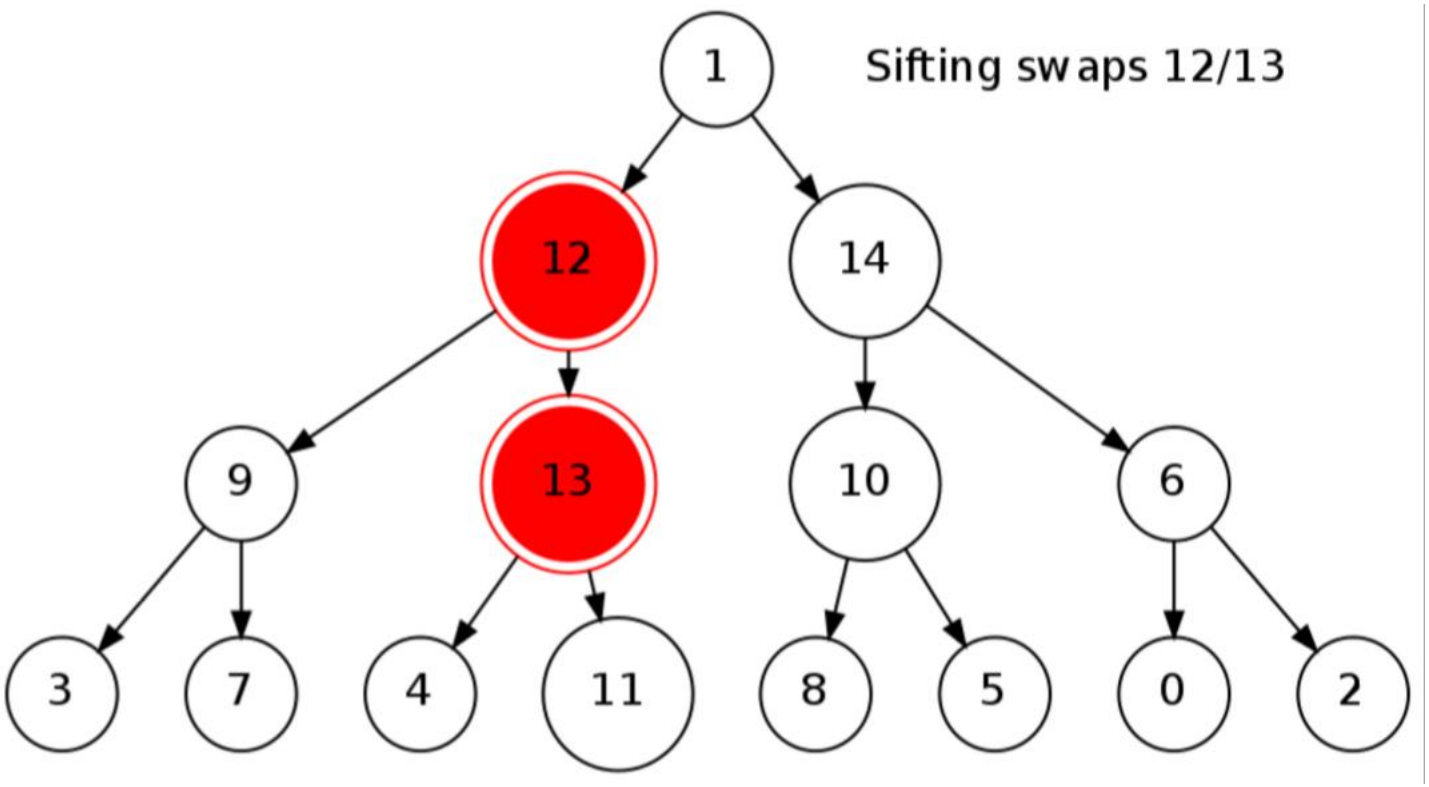
Coursework and Recap

Let's heapify



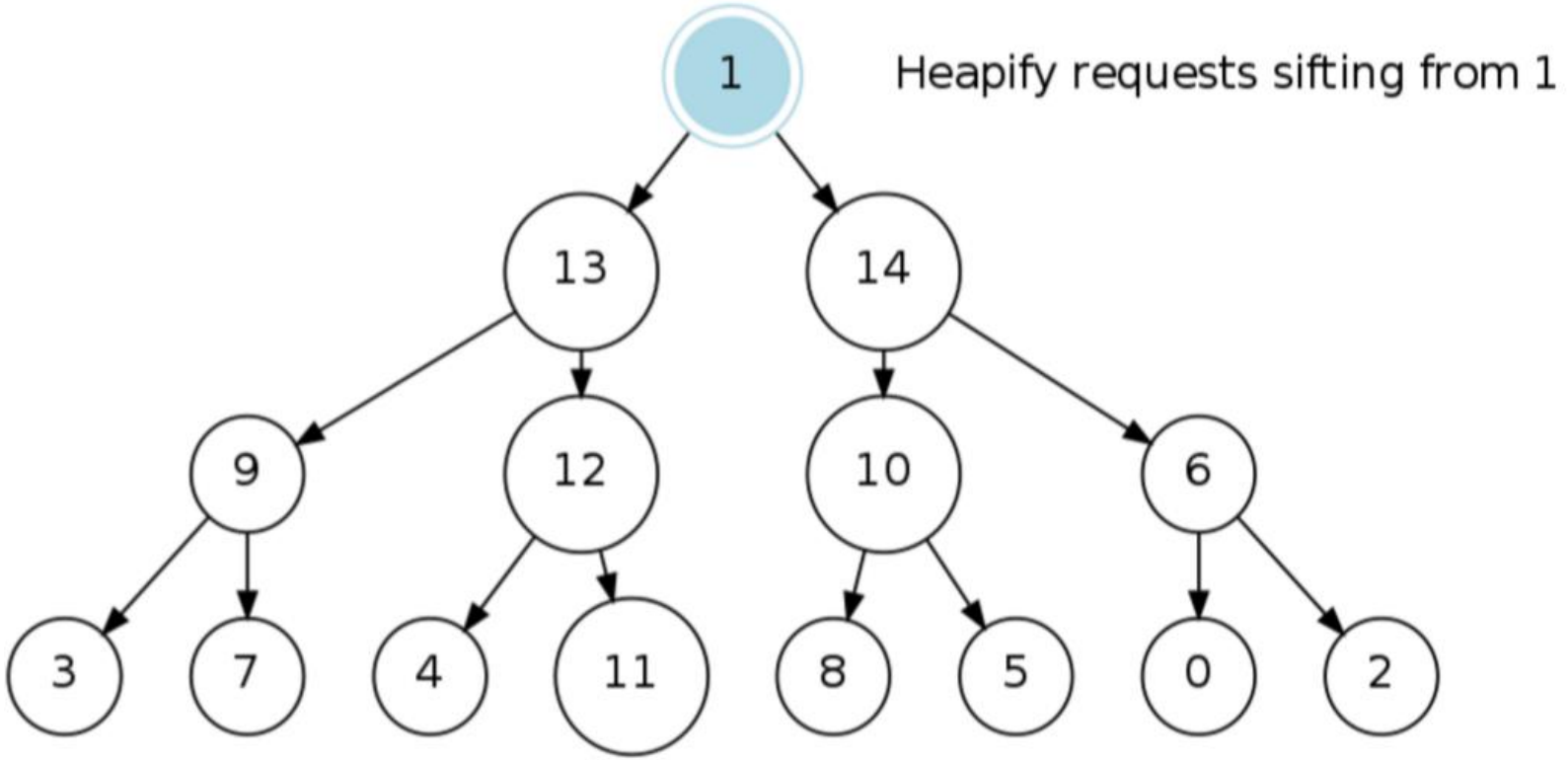
Coursework and Recap

Let's heapify



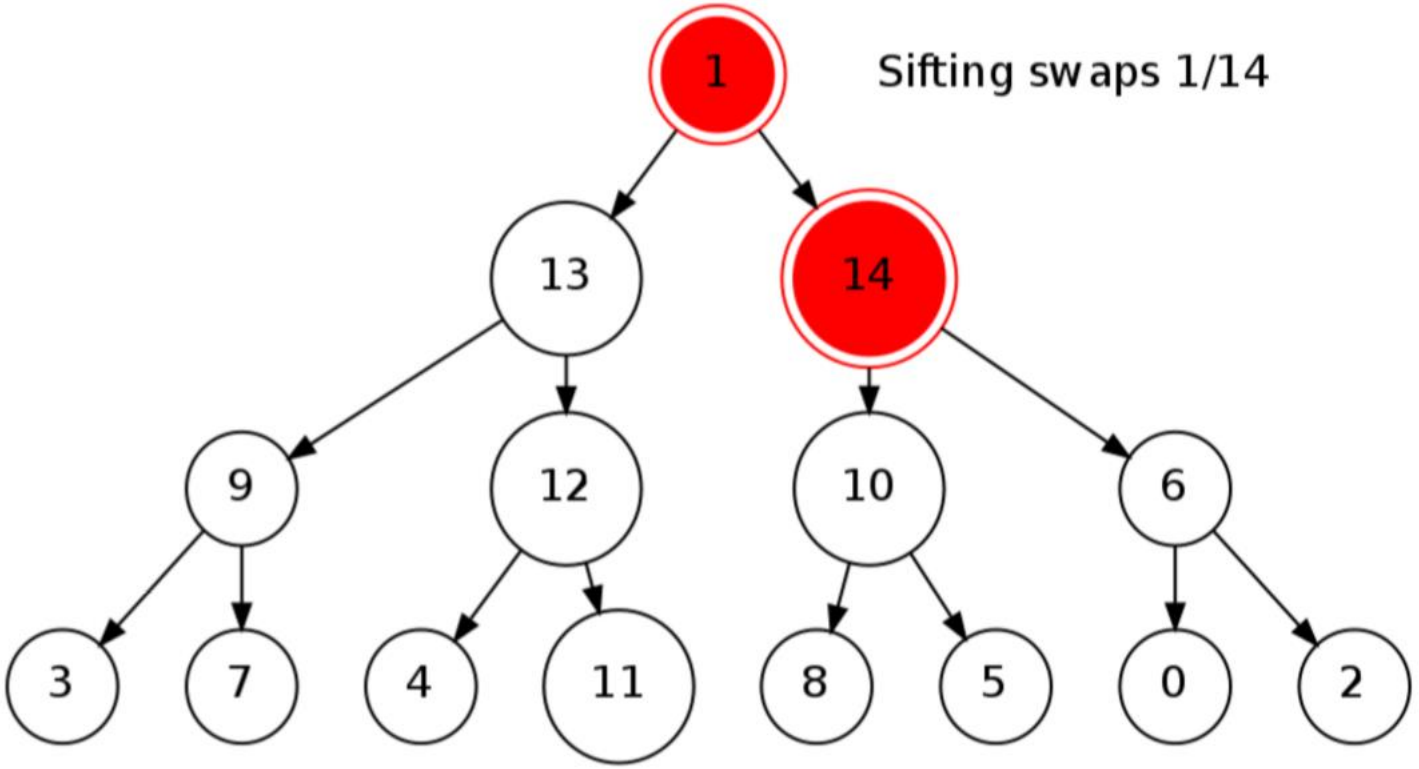
Coursework and Recap

Let's heapify



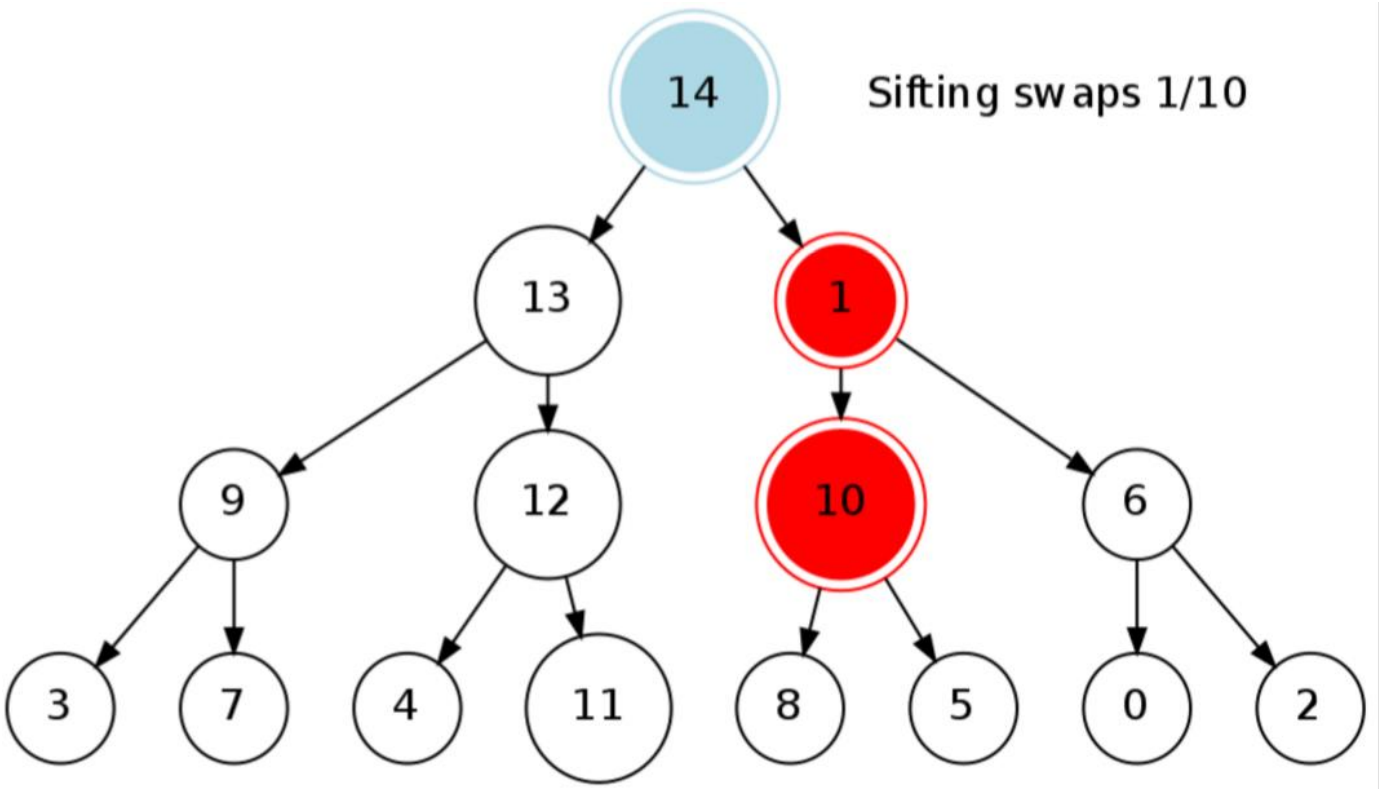
Coursework and Recap

Let's heapify



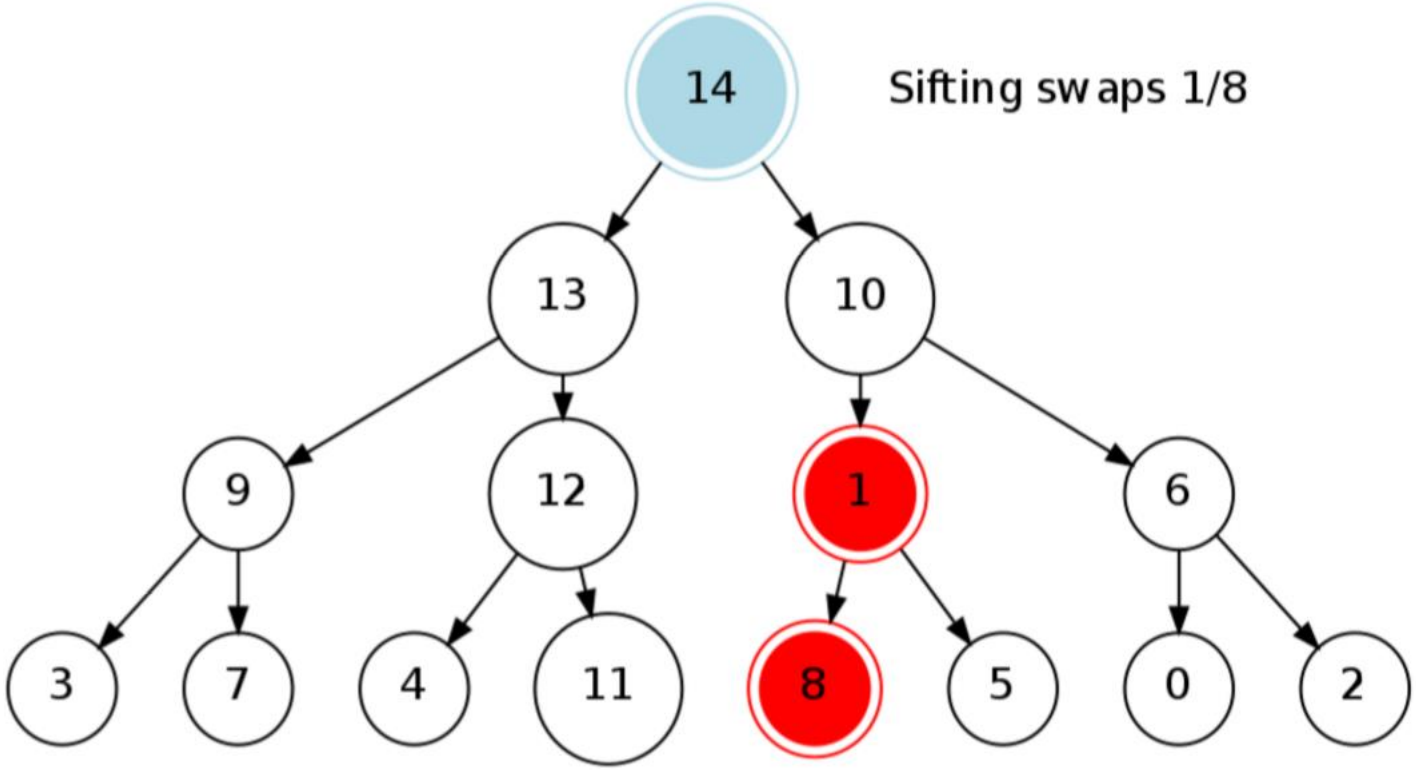
Coursework and Recap

Let's heapify



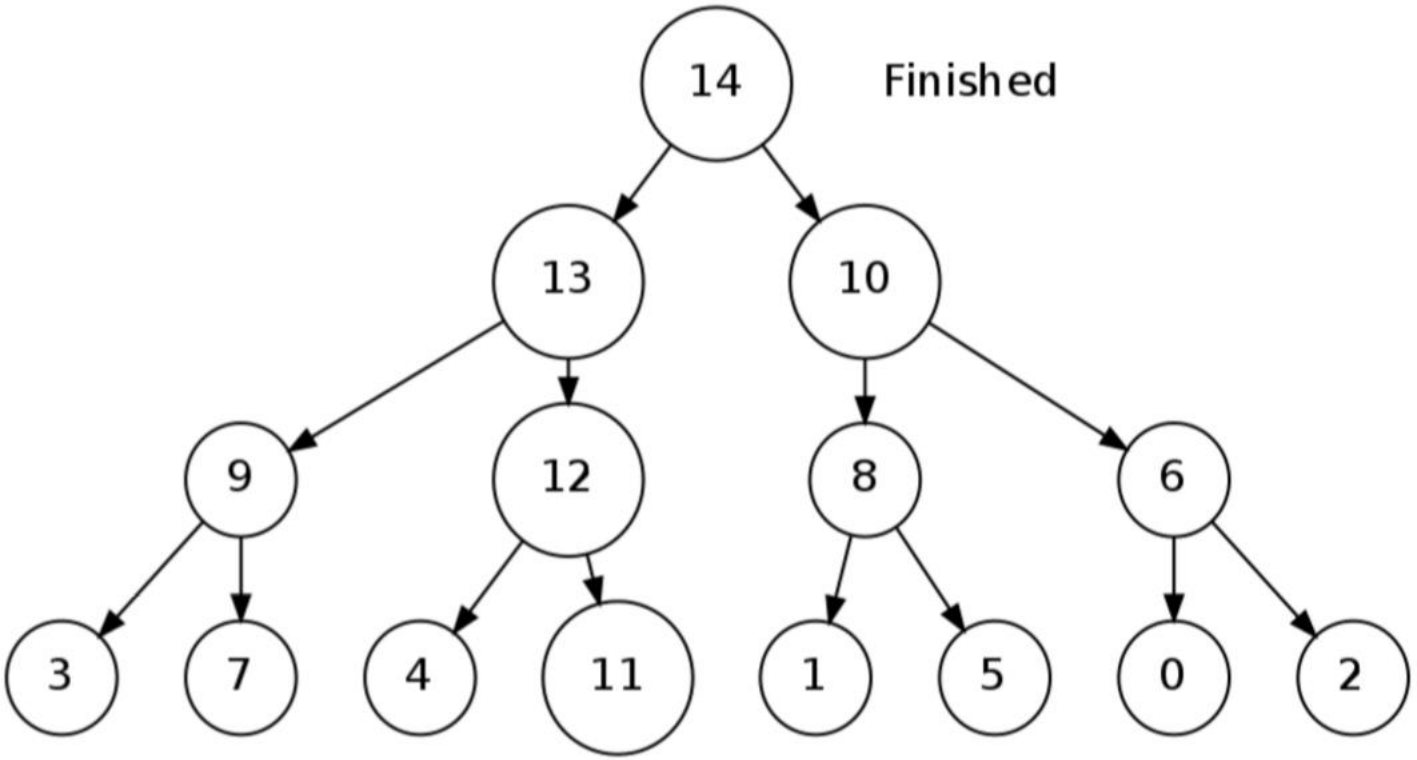
Coursework and Recap

Let's heapify



Coursework and Recap

Let's heapify



Stacks

What are they?

- Simple data structures in which only a specific element can be removed at any given time: the one on the top.
- It is the most recent element added.
- For this reason, a stack is called last in-first out (LIFO) structure.
- Items can only be added to the top.
  - called a push operation
- Items can also only be removed from the top.
  - called a pop operation.

Stacks

- Untitled artwork by Robert Therrien, held by the Tate Gallery.
- True story!
- You can take the plate from the top.
- You can add a plate.
- You can't take a plate from the middle.

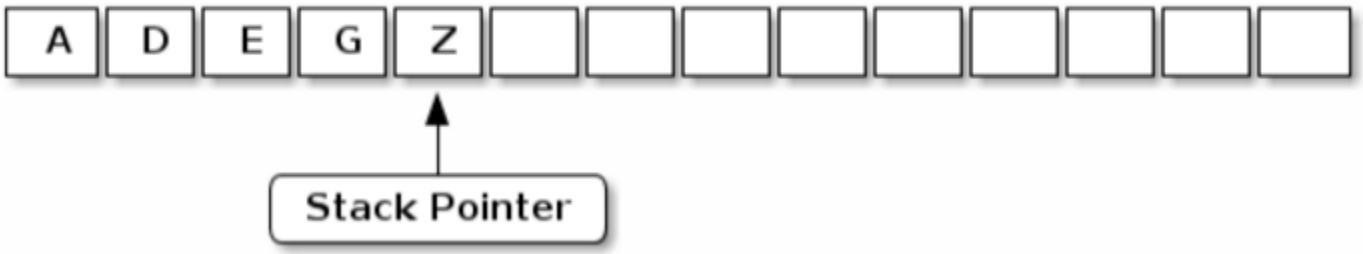
Stacks

Implementing the stack in an array

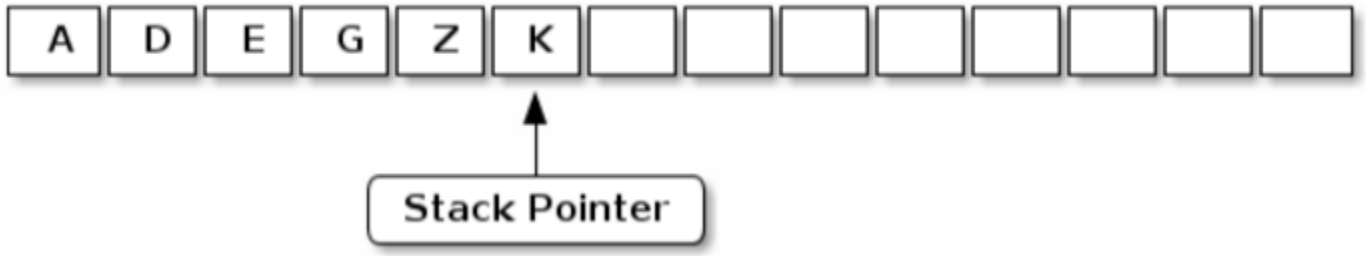
- Stack pointer:
  - This pointer is used to access the items on the stack.
  - The stack maintains a pointer to its topmost element.
  - If an item is pushed, first the stack pointer is incremented, then the item is added to the new position.
  - If an item is popped, the item is returned to the program, then the stack pointer is decremented.

Stacks

Example stack in an array - PUSH

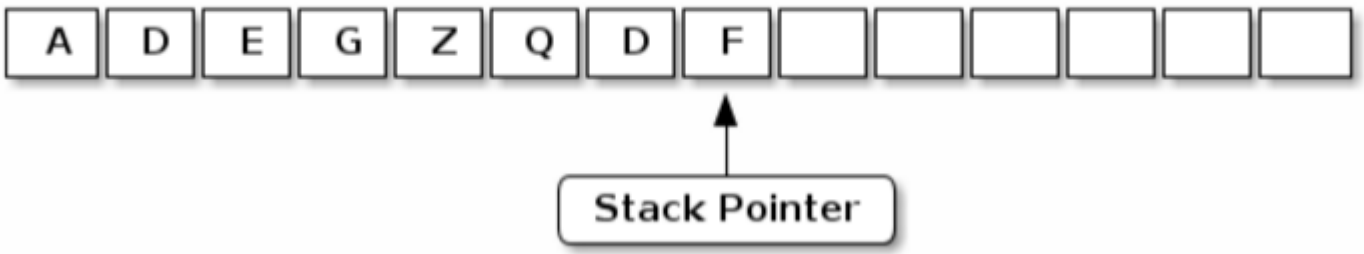


- Now push "K" onto the stack:

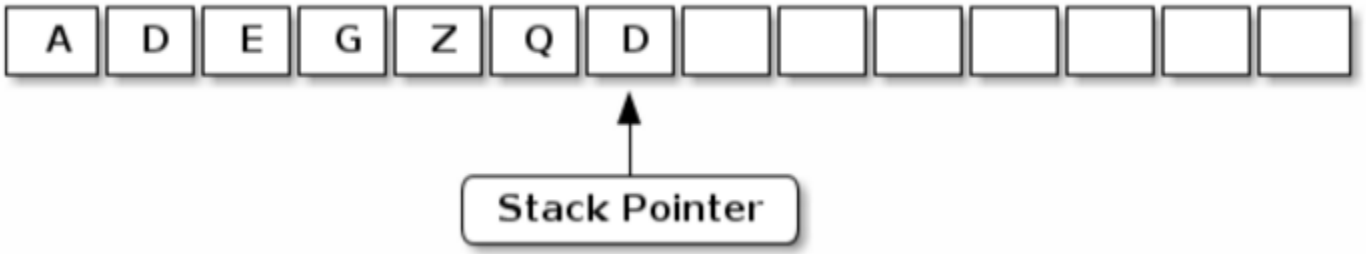


Stacks

Example stack in an array - POP



- A pop operation will result in the letter F being returned, and the stack becomes….



Stack in an array implementation

- A stack can be implemented with an array to store the items.
- An integer can be used for the stack pointer.
- A push() method to add an item an increment the pointer.
- A pop() method to return an item and decrement the pointer.
- An empty() method to indicate when the stack is empty.
- These stack methods implemented like this all have a runtime of O(1).

Stacks

Pseudocode

```
PUSH(S, x)
#Place the item x on the stack
S.top ← S.top + 1
S[S.top] ← x

POP(S)
#Take an item off the stack
S.top = S.top - 1
return S[S.top+1]
```

```
EMPTY(S)
#Check the stack is empty
if(S.top = 0)
    return TRUE
else
    return FALSE
```

Stacks In C++

```
#include <iostream>

int stack[100] ;
int top=0;

void push(int val){
  top++;
  stack[top]=val;
}
```

```
int pop(){
  top--;
  return stack[top+1];
}

bool empty(){
  return top==0;
}
```

```
int main(int argc, char *argv[])
{
  push(10);
  push(100);
  push(123);
  push(9);
  while(!empty())
    std::cout << "Pop gives: "<< pop() << std::endl;
  return 0;
}
```

Stacks

In C++

- Pop gives: 9
- Pop gives: 123
- Pop gives: 100
- Pop gives: 10

Stacks

In Python

```
class Stack(object):
    def __init__(self):
        self.stack=[0]*100
        self.top=0
    def push(self, val):
        self.top+=1
        if self.top>=len(self.stack):
            self.stack=self.stack+[0]*len(self.stack)
        self.stack[self.top]=val
    def pop(self):
        self.top-=1
        return self.stack[self.top+1]
    def empty(self):
        return self.top==0

#Testing
a=Stack()
a.push(10)
a.push(100)
a.push(123)
a.push(9)
while not a.empty():
    print "Pop gives:",a.pop()
```

Stacks

In Python

- Pop gives: 9
- Pop gives: 123
- Pop gives: 100
- Pop gives: 10

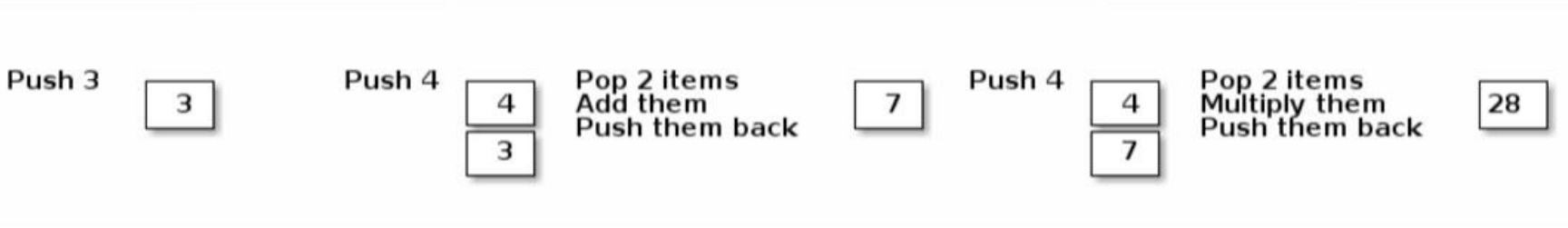
Stacks in the real world

- Your OS is doing stack operations ALL THE TIME!
  - Every time a new function is called, we jump to a new position in code.
  - And we also have (probably) a new set of variables.
  - How do we know where to go back to?
  - How do we know what values were available in the old function when we get there?
  - Every function call pushes this data on the stack.
  - Finishing a function requires the popping of this information from the stack to get back to where we were.

Stacks

Stacks in the real world

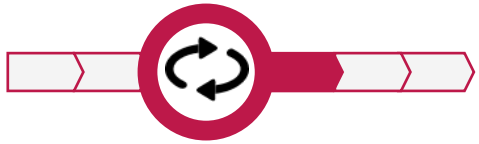
- Evaluating expressions…RPN calculators are a simple example.
- What is 3 4 + 4 *?
- You can implement an RPN calculator by:
  - For very numeric value, push it on the stack.
  - For an operator, pop the operands from the stack, do the operation, and push the result.
  - So the answer to our expression is: ???



Backtracking

- Backtracking
  - Imagine if you had to search a whole directory tree.
    - or navigate a maze
    - or find the best possible outcome of a series of decisions
    - being able to store a point in the process and jump back to it is useful. Having them stored so you can go back multiple levels is even more so.

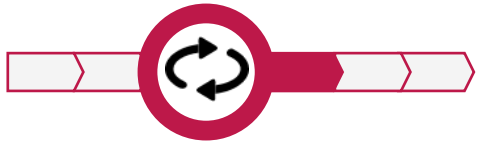
Queues



What are queues?

- A queue is a data structure in which the item which has been in the queue longest is retrieved.
- A queue is called a first in-first out or FIFO structure.
- Like a real-life queue.
- Adding an item to a queue is called an "enqueue" operation.
- Removing an item is a "dequeue“.

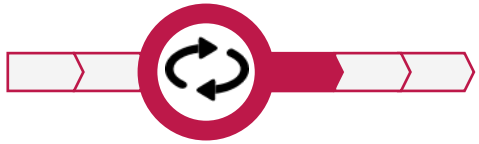
Queues



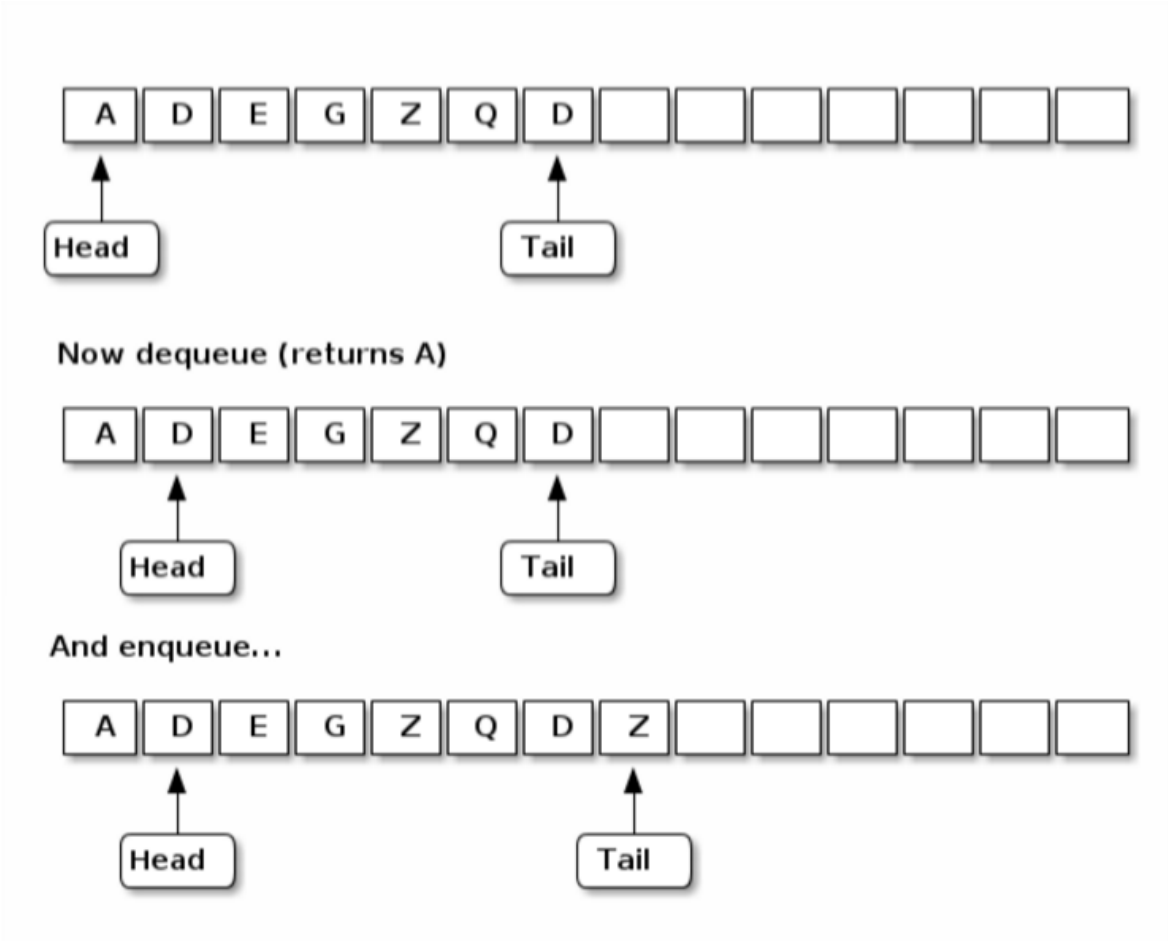
Head and tail

- A queue needs two pointers - one to point to the item at the start (head) of the queue, and one to point at the end (tail).
- It is useful for these pointers to wrap around so the queue uses a fixed amount of memory.
- Enqueue adds a new item to the end of the queue and increments the tail pointer, wrapping if necessary.
- Dequeue removes an item from the front of the queue and decrements the head pointer, wrapping if necessary.

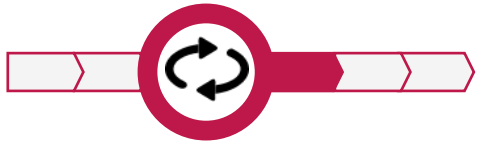
Queues



Example queue



Queues

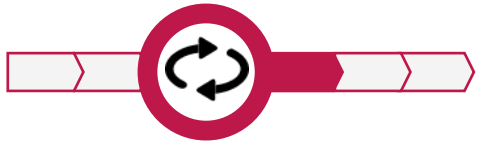


Pseudocode

```
ENQUEUE(Q, x)
Q.tail ← Q.tail + 1
if Q.tail = length[Q]
    Q.tail ← 1
else
    Q.tail ← Q.tail + 1
Q[Q.tail] ← x
```

```
DEQUEUE(Q, x)
x ← Q[Q.head]
if Q.head = length[Q]
    Q.head ← 1
else
    Q.head ← Q.tail + 1
return x
```

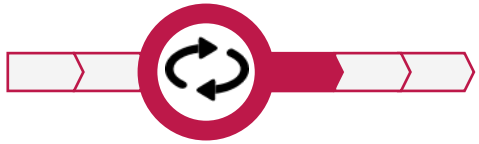
Queues



Queues in the real world

- Pretty obvious.
- Task management.
  - Ensuring item 1 is processed before item 2
- Batch processing.
  - Items in queue A are processed to queue B, etc.
- Cache/buffer.
  - Youtube videos
  - Keyboard input

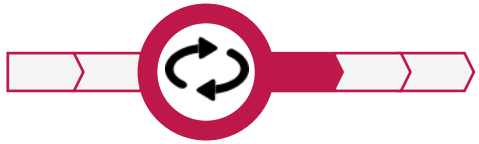
Queues



Actual homework

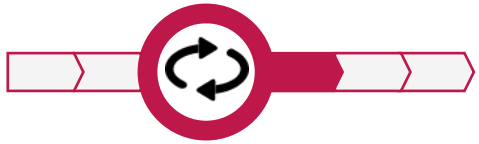
- In your lab session, you will be given a heap worksheet.
- If you didn't get a printed sheet, the electronic version is here: heap.pdf
- Follow the instructions on the sheet.
- For evidence, take a photo of your sheet when complete.
- Array generator

Queues



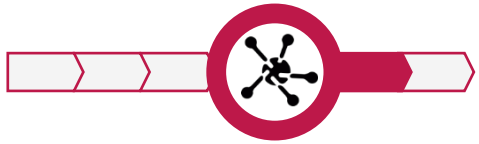
Alternative homework

- Use a stack to implement an RPN calculator.
- This will be useful if you're doing 207SE, too.



BREAK!!!

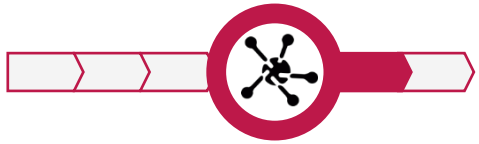
Arrays



The vector - definition

- A method of infection.
  - Nope, that's for viruses
- A point in space.
  - Nope, that's for graphics.
- Direction and magnitude.
  - Still no. Maths!
- A contiguous, ordered set of elements in which each element is referred to by an index, which is a nonnegative integer.
  - Bingo.

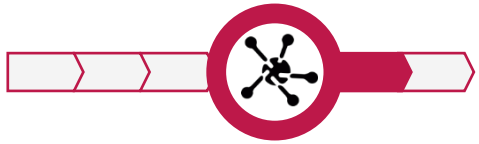
Arrays



The array as a vector

- Many languages have the array as a basic type.
- C++ does, for example.
- In Python, we have the "list" which is almost the same, apart from some convenient methods and "syntactic sugar“.

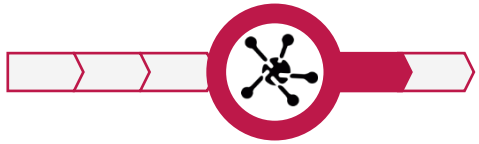
Arrays



What else is a vector?

- Records in a database that have unique consecutive IDs (and aren't ever deleted).
- Books on a shelf (position is the index).
- Other data structures…

Arrays

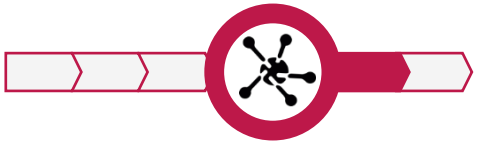


Pointers

- C and C++ have pointers.
- Everyone knows this.

```
int main(int argc, char *argv[])
{

  int a=1;
  int b=a;

  a=2;

  std::cout << b << std::endl;

  return 0;
}
```

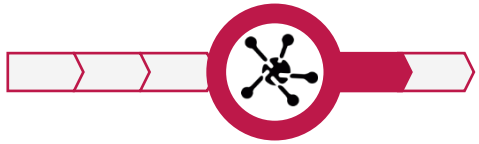
Arrays



Pointers

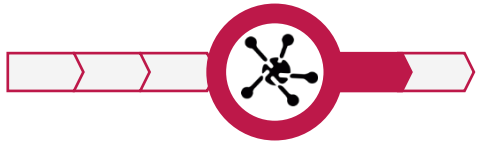
```
int main(int argc, char *argv[])
{

  int* a = new int;
  *a=1;

  int* b;
  b=a;

  *a=2;

  std::cout << *b << std::endl;

  return 0;
}
```

Arrays



Pointers

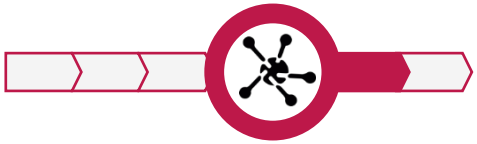
Arrays



Pointers

- http://www.cplusplus.com/doc/tutorial/pointers/

Arrays



Python

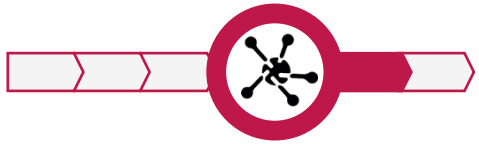
- Pah, Python doesn't require such things, right?

```
a=1
b=a
b=2
print a
```

- And another way…

```
a=[1]
b=a
a[0]=2
print b
```

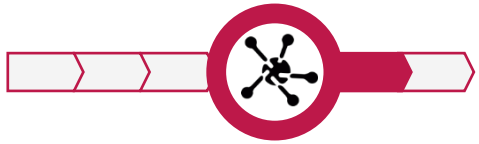
Arrays



Python and pointers

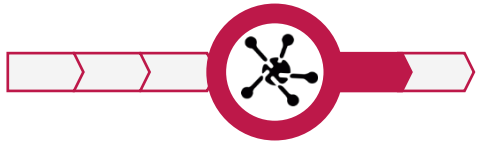
- Just as with C++, basic types are treated differently to objects and arrays.
- Lists are an object, so since we have no special notation (int*in C++, for e.g.) for pointers, this was an easy way to do it.
- Essentially, we are referring to an object that contains a value rather than the value itself.

Arrays



Back to arrays

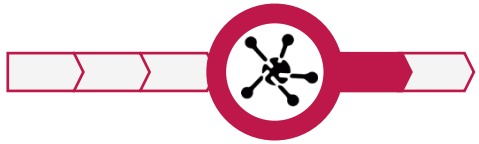
- Accessing items in an array is simple.
- We just use the index.
- In memory, the item we want is at i×k, where i is the index and k is the size of each item.
- This calculation is done for us when we use the array.
- Big O notation for accessing an item???



Back to arrays

- Accessing items in an array is simple.
- We just use the index.
- In memory, the item we want is at i×k, where i is the index and k is the size of each item.
- This calculation is done for us when we use the array.
- Big O notation for accessing an item???
- O(1).

Arrays

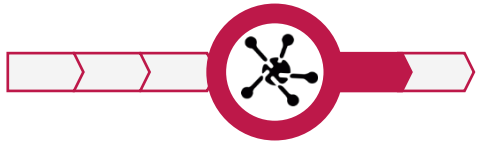


Back to arrays

```
#include <iostream>
int main(int argc, char *argv[])
{
  int a[5];
  a[0]=11;    a[1]=22;    a[2]=33;    a[3]=44;    a[4]=55;

  std::cout << "Address of a[0]: " << a << std::endl;
  std::cout << "Address of a[1]: " << &a[1] << std::endl;
  std::cout << "Integer size: " << sizeof(int) << std::endl;
  return 0;
}
```

```
Address of a[0]: 0x7fffaa7f99b0
Address of a[1]: 0x7fffaa7f99b4
Integer size: 4
```

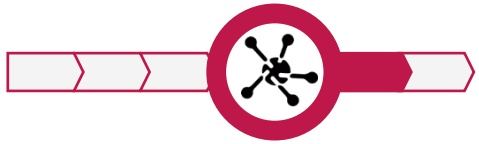
Arrays



Insert or delete an item

- To insert a new item in an array, not overwriting a value, is more difficult.
  - Each item after the insertion point needs to be shifted.
- Big O notation = ??

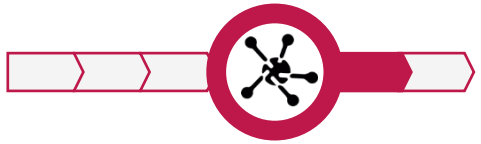
Arrays



Insert or delete an item

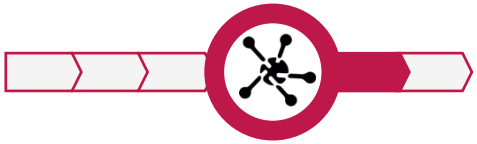
- To insert a new item in an array, not overwriting a value, is more difficult.
  - Each item after the insertion point needs to be shifted.
- This is O(n).

Arrays



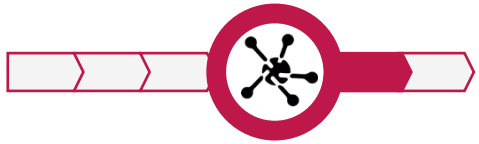
```
#include <iostream>

void insert(int ar[], int len, int val, int idx){
  for(int i=len-1; i>idx; i--) //Last item is lost
    ar[i]=ar[i-1];
  ar[idx]=val;
}

int main(int argc, char *argv[])
{
  int a[10];
  a[0]=11;    a[1]=22;    a[2]=33;    a[3]=44;    a[4]=55;
  a[5]=0;     a[6]=0;     a[7]=0;     a[8]=0;     a[9]=0;
  insert(a,10,99,3);
  for(int i=0;i<10;i++)
    std::cout << i<<": " << a[i] << std::endl;
  return 0;
}
```

Arrays



```
0: 11
1: 22
2: 33
3: 99
4: 44
5: 55
6: 0
7: 0
8: 0
9: 0
```

Arrays

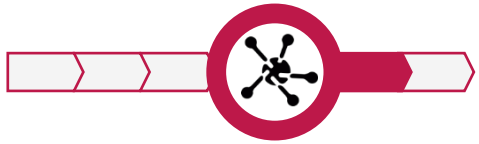


And in Python…

```
a=[11,22,33,44,55]
a.insert(3,99)
print a
```

```
[11, 22, 33, 99, 44, 55]
```

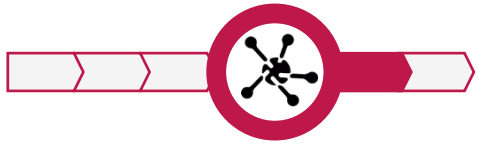
Arrays



And in Python…

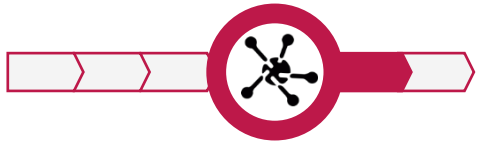
- Only 1 operation, so O(1), right?

Arrays



And in Python…

- Only 1 operation, so O(1), right?
- NO!
- We only have one line of code, but the syntax is simply hiding what is really happening.
- Underneath, Python lists are made of arrays.



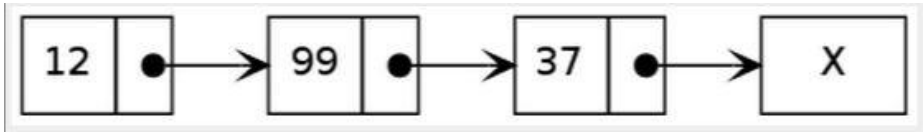
Delete

- Same problem in reverse operation.

Linked Lists

Linked Lists

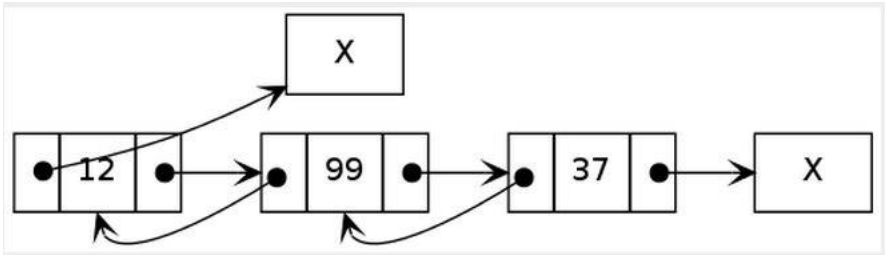
- A linked list is a data structure which also stores items in order, accessed by index, without gaps.
- The items have contiguous indices, but it does not use contiguous memory - an element can be stored anywhere in memory, regardless of where the element with the index one higher or one lower is stored.
- Each element stores a pointer to the next element - the "link“.
- Last element points to null.
- First item is the "head“.
- Access the ith element by following i links from the head.



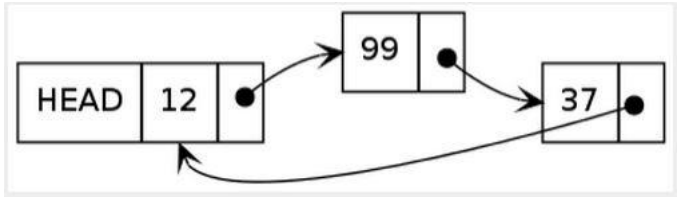
Linked Lists

Double linked lists and circular lists

- A doubly linked list also stores a pointer to the previous element of the list.



- A circular linked list stores a pointer from the last element to the first element.



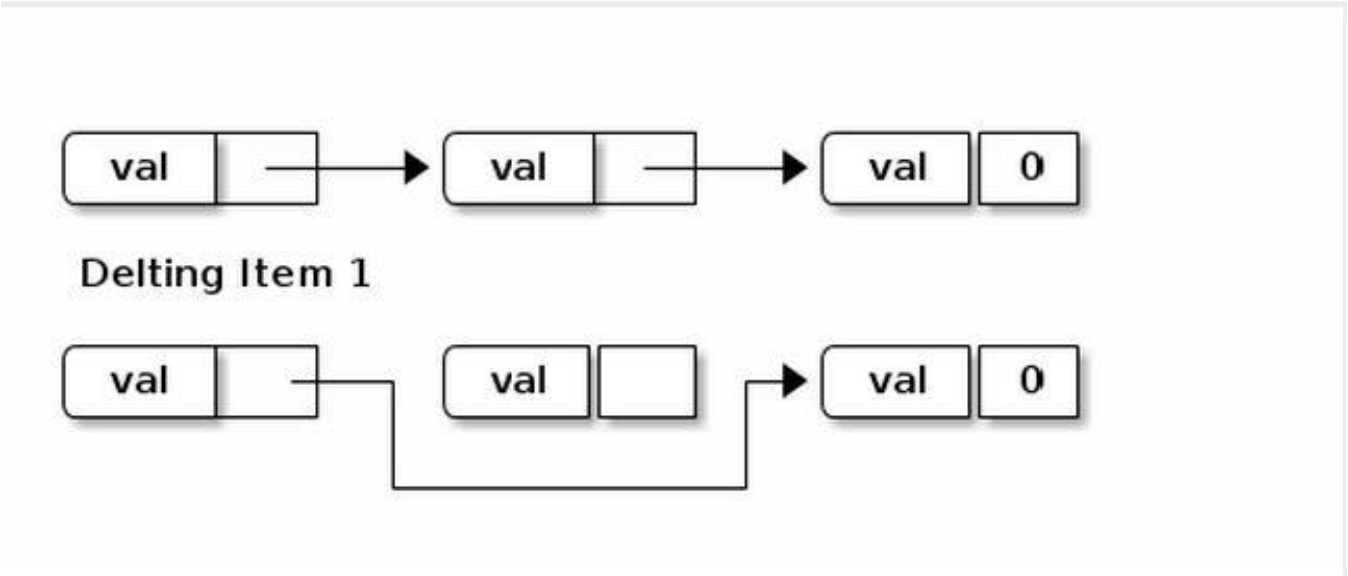
Linked Lists

Accessing and changing content

- To reach a given item, the list is traversed, beginning with the head.
  - Is this the item you want?
  - If not, follow the link and repeat.
- Deleting an item is achieved by changing the link from the previous item to point to the one **after** the one we want deleted.
- Inserting is done by creating a new item and then changing the link from the item we want it to follow to point to it, and making the new item's link point to the item that previously followed the item that now links to it.
- These operations make more sense with a diagram and some notation.

Linked Lists

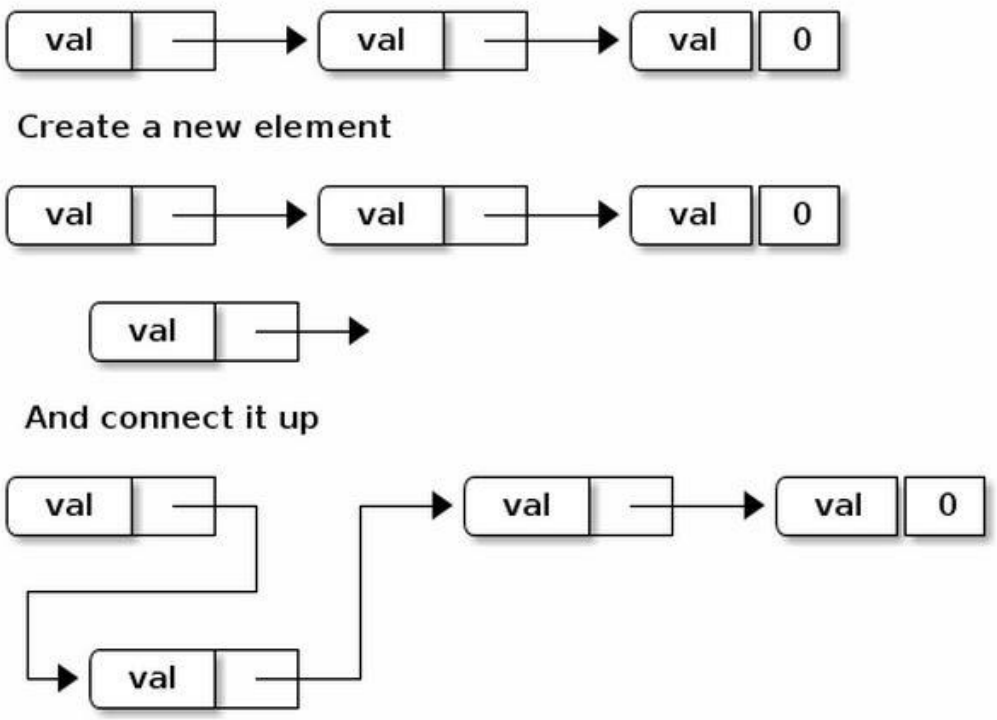
Deletion



- If your language doesn't perform automatic garbage collection, you must now erase/delete the disconnected element.

Linked Lists

Insertion



Linked Lists

Efficiency

- Linked lists have the opposite characteristics of an array.
- Access to a particular element is slow because we have to work our way through each element of the list to get there, it's O(n).
- However, insertion and deletion is fast (once we are at the element we want to insert at) because we merely have to change which element the pointer is pointing to, this is O(1).

Linked Lists

Creating a linked list

- We can consider a doubly linked list nodeN to have three attributes:
  1. **value[N] -** the data item stored by the node.
  2. **previous[N] -** a pointer to the previous node in the list.
  3. **next[N] –** a pointer to the next node in the list.
- We can also define a list object, L, which contains two attributes:
  1. **head[L] -** the first node in the list.
  2. **tail[L] -** the last node in the list.
- This is common pseudocode style notation.

Linked Lists

Creating a linked list

- We can consider a doubly linked list nodeN to have three attributes:
  - **N.value -** the data item stored by the node.
  - **N.previous -** a pointer to the previous node in the list.
  - **N.next -** a pointer to the next node in the list.
- We can also define a list object, L, which contains two attributes:
  - **L.head -** the first node in the list.
  - **L.tail -** the last node in the list.

Linked Lists

Creating a linked list

- What is the difference?
  - Nothing, just more syntactic sugar
- In the definitions of object functions, the first parameter is still listed as the object in question:
  - Like "self" in Python.
  - Like the implicit "this" in C++.

Linked Lists

Operation INSERT

- We can define an operation insert which will place the node X immediately after the node N in the list L using the following pseudocode, for the first item we can just use
  N = Ø (A NULL pointer).
- Note that N is a node, not an index, just like X.

Linked Lists

Operation INSERT

```
LIST-INSERT(L, N, X)
if N ≠ ∅
    X.next ← N.next
    N.next ← X
    X.prev ← N
    if X.next ≠ ∅
        X.next.prev ← x
if L.head = ∅
    L.head ← L.tail ← X
    X.prev ← X.next ← ∅
else if L.tail = N
    L.tail ← X
```

Linked Lists

Operation REMOVE

```
LIST-REMOVE(L, N)
if N.prev ≠ Ø
    N.prev.next ← N.next
else
    L.head ← N.next
if N.next ≠ Ø
    N.next.prev ← N.prev
else
    L.tail ← N.prev
```

Linked Lists

C++ implementation

- http://pastebin.com/MzDCFyD2

Python implementation

- http://pastebin.com/Kxe2ZrJt

Linked Lists

In summary

- Arrays are:
  - Very efficient for access and modification of a specific element.
  - Not efficient for insertion and removal.
- Linked lists are:
  - Not efficient for finding, access and modification of a specific element.
  - Very efficient for insertion and removal.

Linked Lists

Pre-homework

- C++ people only this week:
  - create an array of integers, show how you can access every item using array notation and simple pointer arithmetic.
- If those terms don't mean anything to you, look them up. They are important!

Linked Lists

Actual homework

- In your chosen language, implement the node delete function.
- Tips:
  - Extend the code from the lecture.
- Use the code as a template - you can see how to perform all the basic operations, so all you have to do is decide the order of them.

Linked Lists

Alternate homework

- Extend the implementation of a doubly linked list given in the class.
- Decide how best to solve the issue of being unable to prepend.
- Show your solution as pseudocode and then implement it.
- Read this paper
- Select a sorting algorithm (Bubble is fine) and implement it such that it operates on a doubly-linked list.

Discussion