



CS 225

Data Structures

October 25 – Hashing Analysis

G Carl Evans

Running Times

The expected number of probes for find(key) under SUHA

Linear Probing:

- Successful: $\frac{1}{2}(1 + 1/(1-\alpha))$
- Unsuccessful: $\frac{1}{2}(1 + 1/(1-\alpha))^2$

(Don't memorize these equations, no need.)

Double Hashing:

- Successful: $1/\alpha * \ln(1/(1-\alpha))$
- Unsuccessful: $1/(1-\alpha)$

Instead, observe:

- **As α increases:**

Separate Chaining:

- Successful: $1 + \alpha/2$
- Unsuccessful: $1 + \alpha$

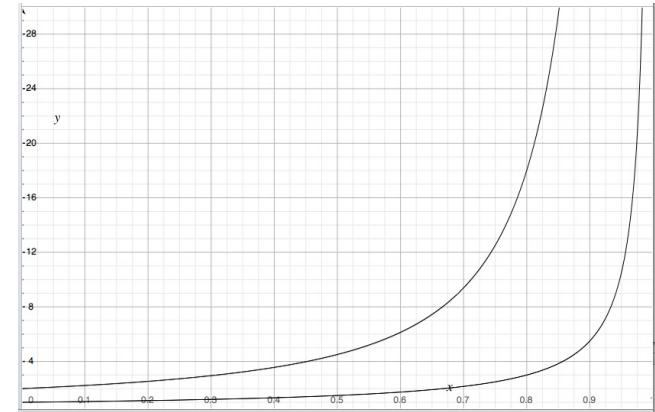
- **If α is constant:**

Running Times

The expected number of probes for find(key) under SUHA

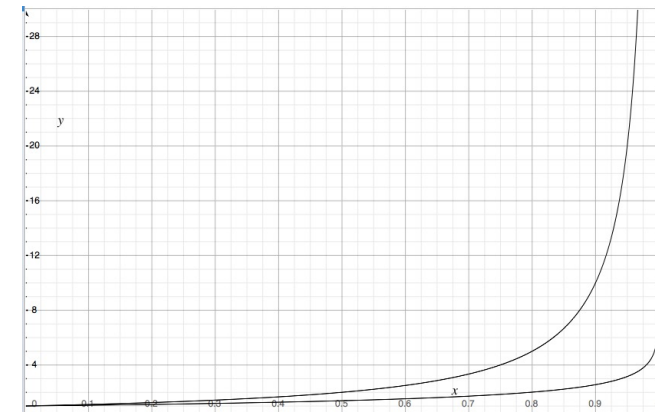
Linear Probing:

- Successful: $\frac{1}{2}(1 + 1/(1-\alpha))$
- Unsuccessful: $\frac{1}{2}(1 + 1/(1-\alpha))^2$



Double Hashing:

- Successful: $1/\alpha * \ln(1/(1-\alpha))$
- Unsuccessful: $1/(1-\alpha)$





ReHashing

What if the array fills?





Which collision resolution strategy is better?

- Big Records:
- Structure Speed:

What structure do hash tables replace?

What constraint exists on hashing that doesn't exist with BSTs?

Why talk about BSTs at all?

Running Times

	Hash Table	AVL	Linked List
Find	SUHA: Worst Case:		
Insert	SUHA: Worst Case:		
Storage Space			



std data structures

std::map

::operator[]

::insert

::erase

::lower_bound(key) → Iterator to first element \leq key

::upper_bound(key) → Iterator to first element $>$ key



std data structures

std::unordered_map

`::operator[]`

`::insert`

`::erase`

~~`::lower_bound(key)` → Iterator to first element \leq key~~

~~`::upper_bound(key)` → Iterator to first element $>$ key~~

std data structures

std::unordered_map

`::operator[]`

`::insert`

`::erase`

~~`::lower_bound(key)` → Iterator to first element \leq key~~

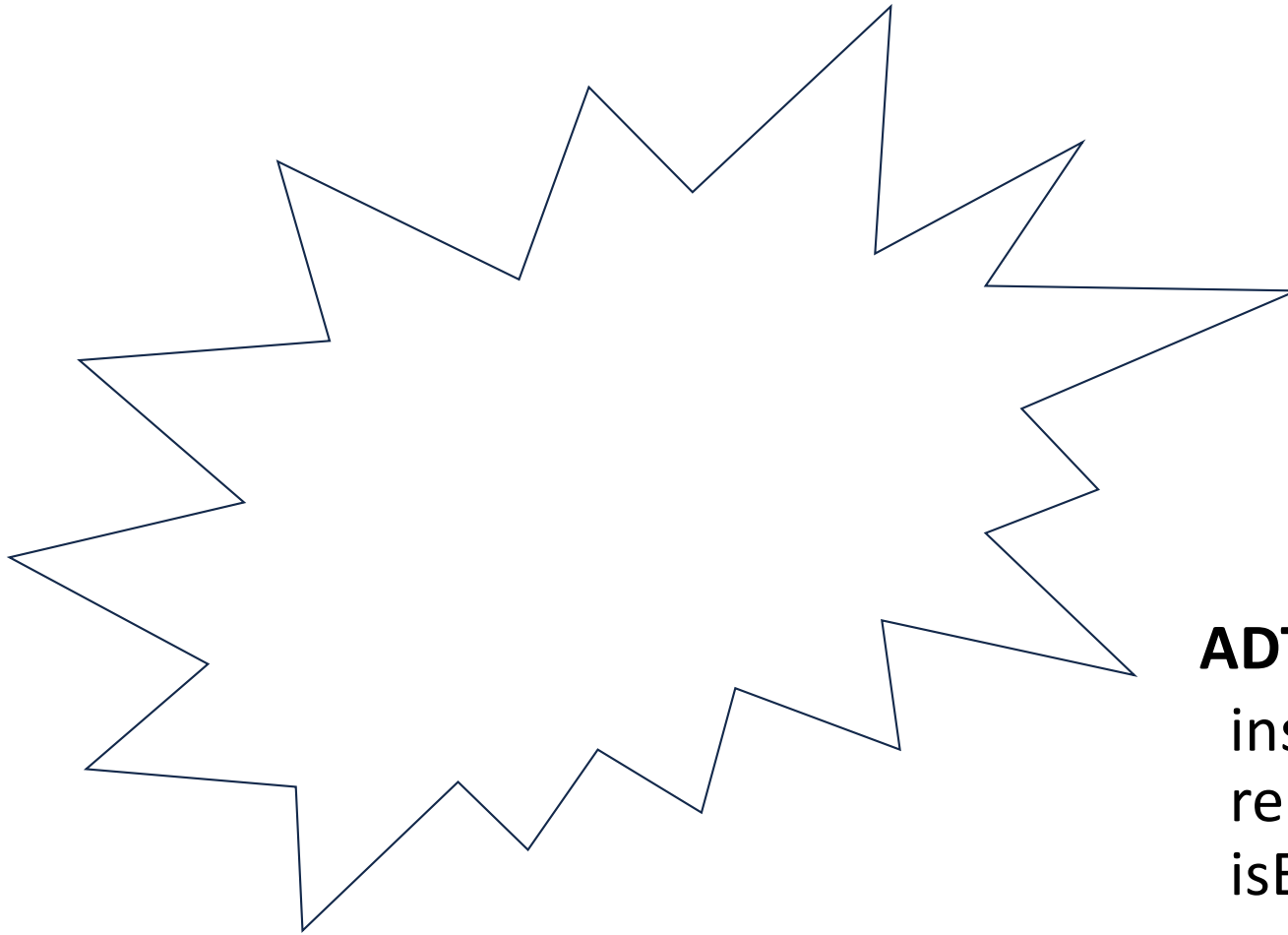
~~`::upper_bound(key)` → Iterator to first element $>$ key~~

`::load_factor()`

`::max_load_factor(ml)` → Sets the max load factor



Secret, Mystery Data Structure

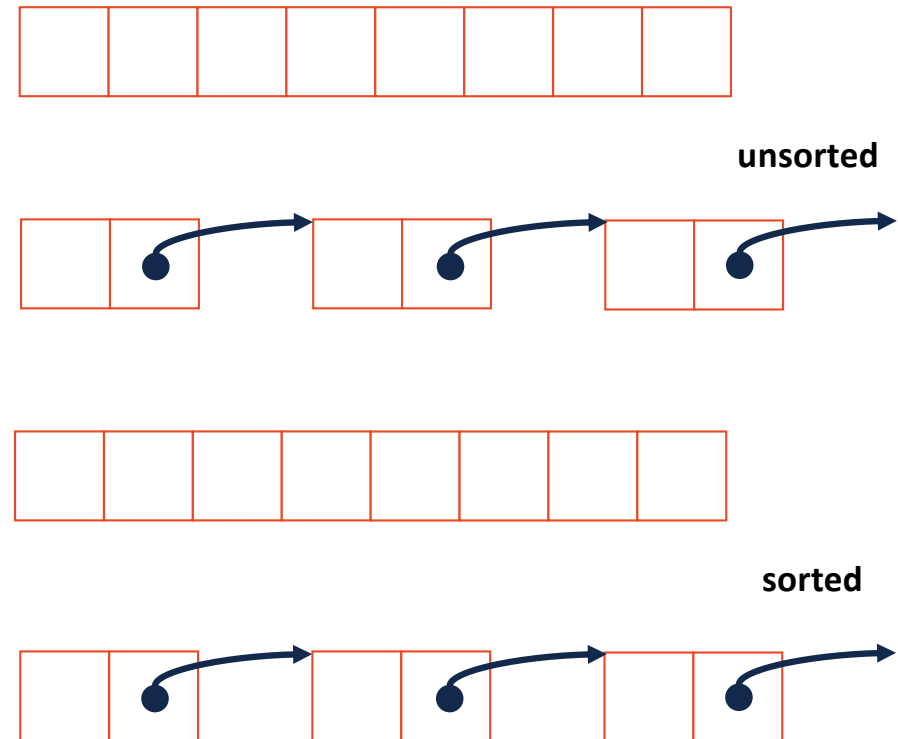


ADT:

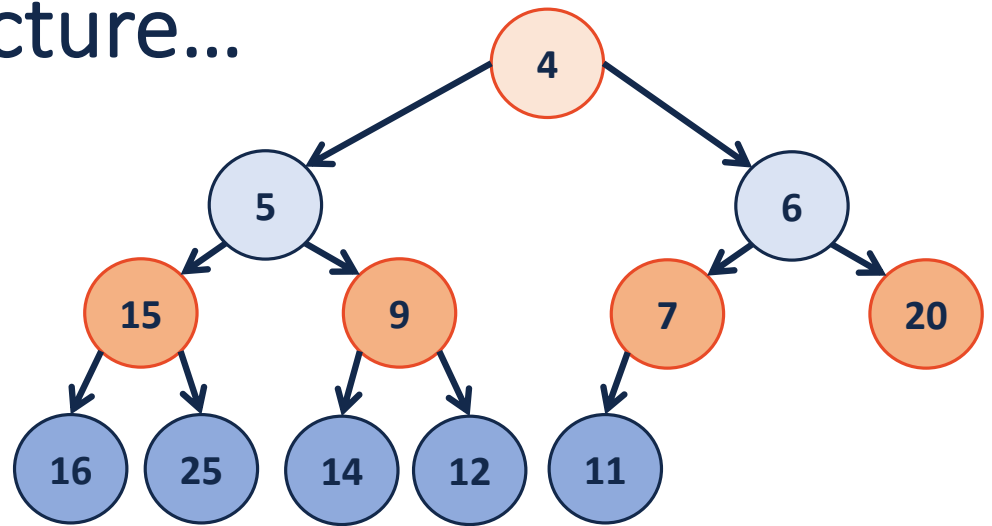
insert
remove
isEmpty

Priority Queue Implementation

insert	removeMin
$O(n)$	$O(n)$
$O(1)$	$O(n)$
$O(\lg(n))$	$O(1)$
$O(\lg(n))$	$O(1)$



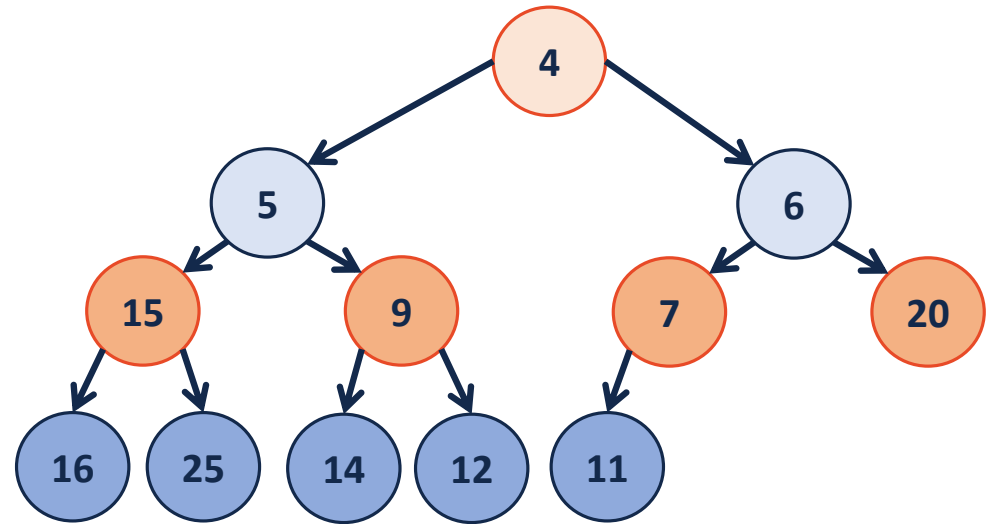
Another possibly structure...



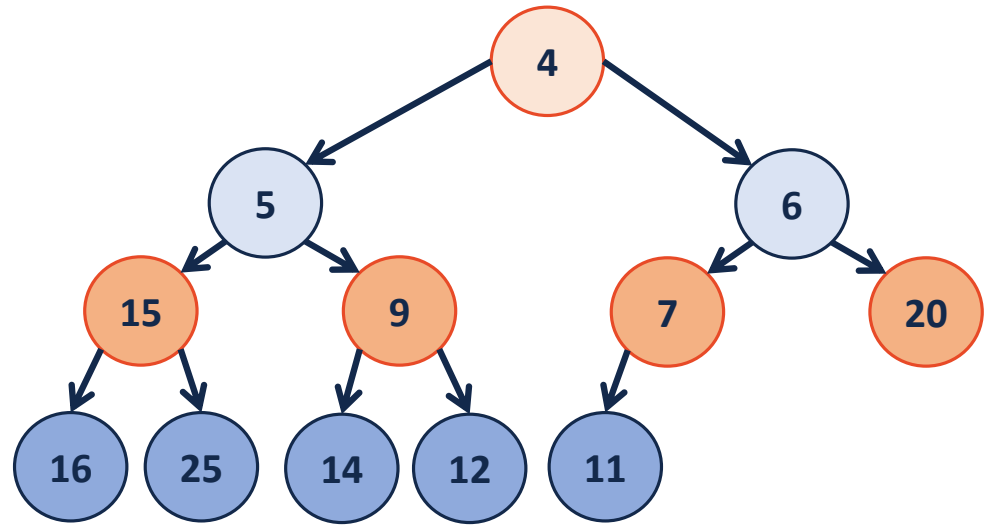
(min)Heap

A complete binary tree T is a min-heap if:

- $T = \{\}$ or
- $T = \{r, T_L, T_R\}$, where r is less than the roots of $\{T_L, T_R\}$ and $\{T_L, T_R\}$ are min-heaps.

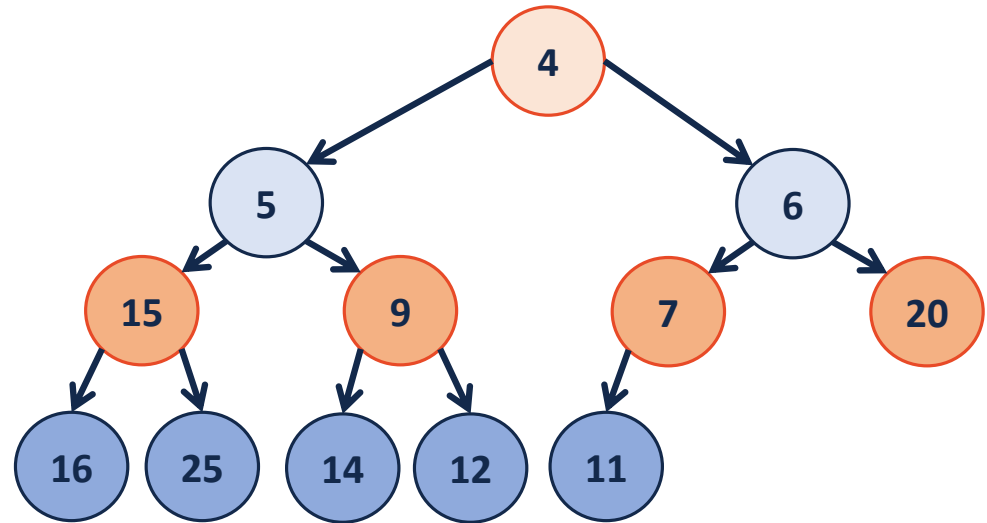


(min)Heap



4	5	6	15	9	7	20	16	25	14	12	11			
---	---	---	----	---	---	----	----	----	----	----	----	--	--	--

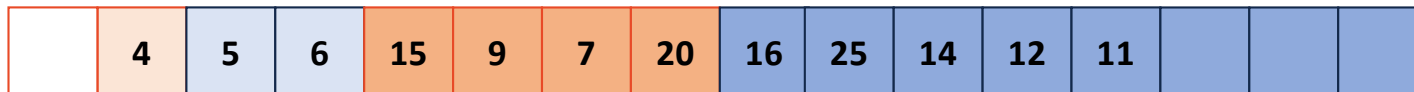
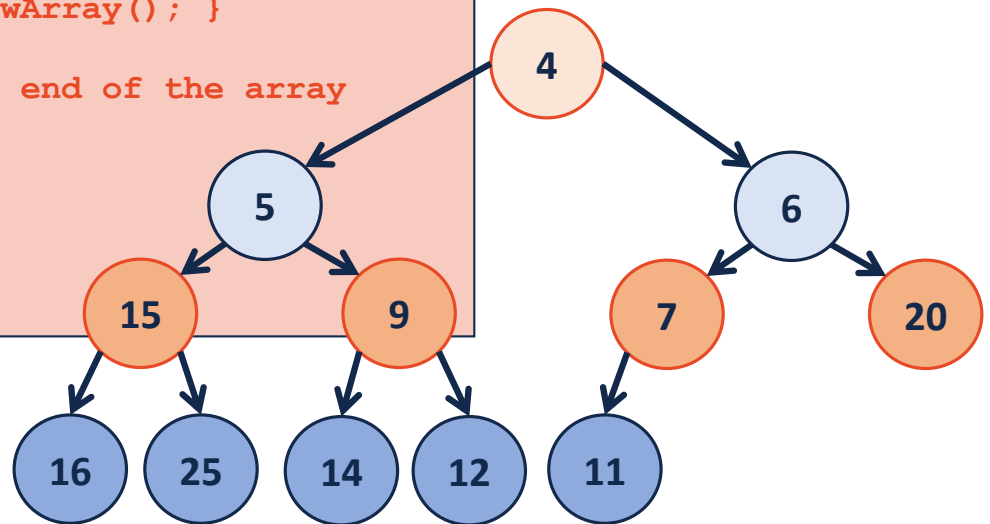
insert



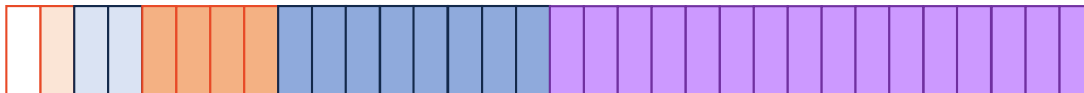
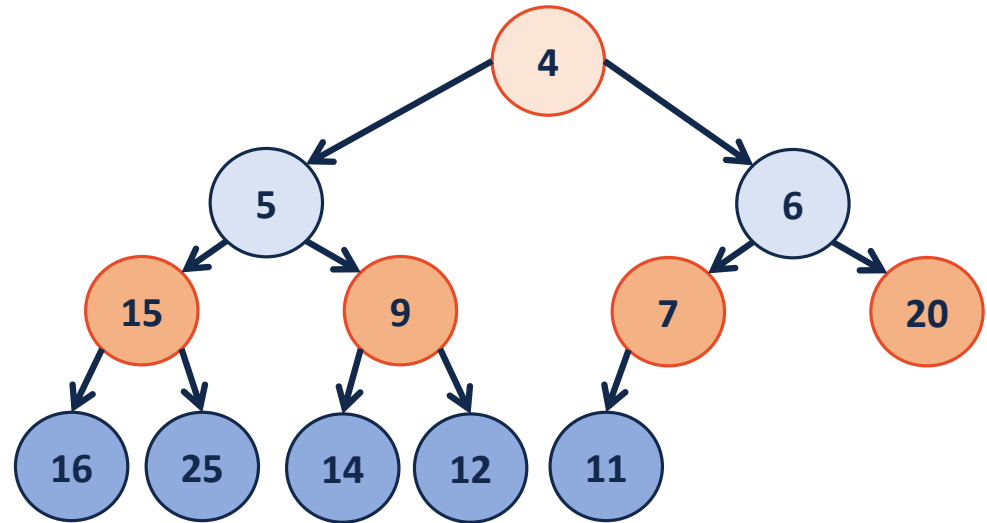
	4	5	6	15	9	7	20	16	25	14	12	11			
--	---	---	---	----	---	---	----	----	----	----	----	----	--	--	--

insert

```
1 template <class T>
2 void Heap<T>::_insert(const T & key) {
3     // Check to ensure there's space to insert an element
4     // ...if not, grow the array
5     if ( size_ == capacity_ ) { _growArray(); }
6
7     // Insert the new element at the end of the array
8     item_[++size] = key;
9
10    // Restore the heap property
11    _heapifyUp(size);
12 }
```



growArray

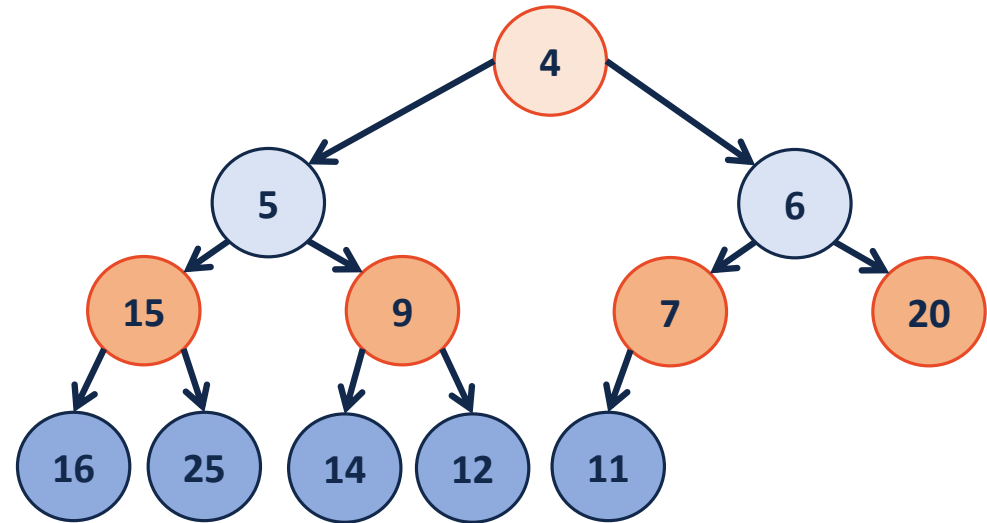


insert - heapifyUp

```
1 template <class T>
2 void Heap<T>::_insert(const T & key) {
3     // Check to ensure there's space to insert an element
4     // ...if not, grow the array
5     if ( size_ == capacity_ ) { _growArray(); }
6
7     // Insert the new element at the end of the array
8     item_[++size] = key;
9
10    // Restore the heap property
11    _heapifyUp(size);
12 }
```

```
1 template <class T>
2 void Heap<T>::_heapifyUp( _____ ) {
3     if ( index > _____ ) {
4         if ( item_[index] < item_[ parent(index) ] ) {
5             std::swap( item_[index], item_[ parent(index) ] );
6             _heapifyUp( _____ );
7         }
8     }
9 }
```

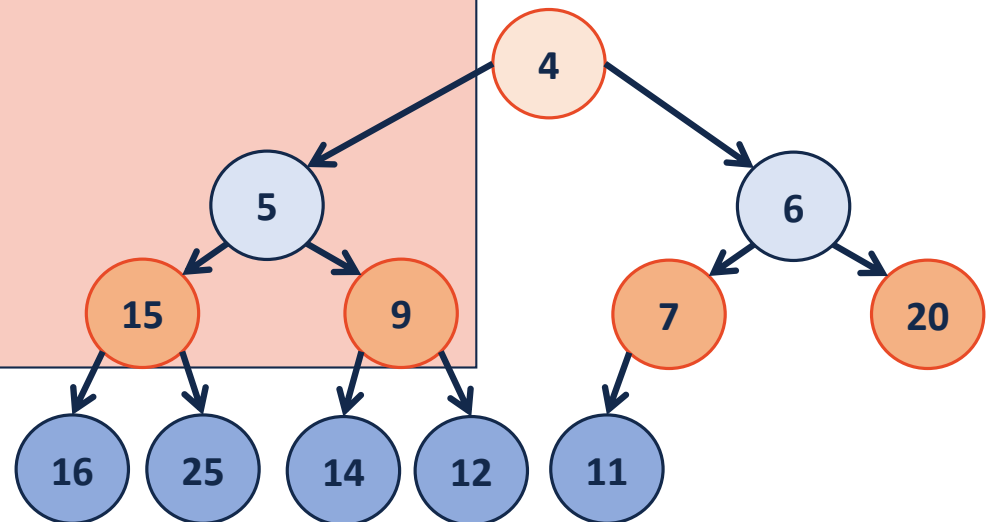
removeMin



	4	5	6	15	9	7	20	16	25	14	12	11			
--	---	---	---	----	---	---	----	----	----	----	----	----	--	--	--

removeMin

```
1 template <class T>
2 void Heap<T>::_removeMin() {
3     // Swap with the last value
4     T minValue = item_[1];
5     item_[1] = item_[size_];
6     size--;
7
8     // Restore the heap property
9     heapifyDown();
10
11    // Return the minimum value
12    return minValue;
13 }
```



removeMin - heapifyDown

```
1  template <class T>
2  void Heap<T>::_removeMin() {
3      // Swap with the last value
4      T minValue = item_[1];
5      item_[1] = item_[size_];
6      size--;
7
8      // Restore the heap property
9      _heapifyDown();
10
11     // Return the minimum value
12     return minValue;
13 }
```

```
1  template <class T>
2  void Heap<T>::_heapifyDown(int index) {
3      if ( !_isLeaf(index) ) {
4          T minChildIndex = _minChild(index);
5          if ( item_[index] > item_[minChildIndex] ) {
6              std::swap( item_[index], item_[minChildIndex] );
7              _heapifyDown( minChildIndex );
8          }
9      }
10 }
```

Array Abstractions

