



CS 225

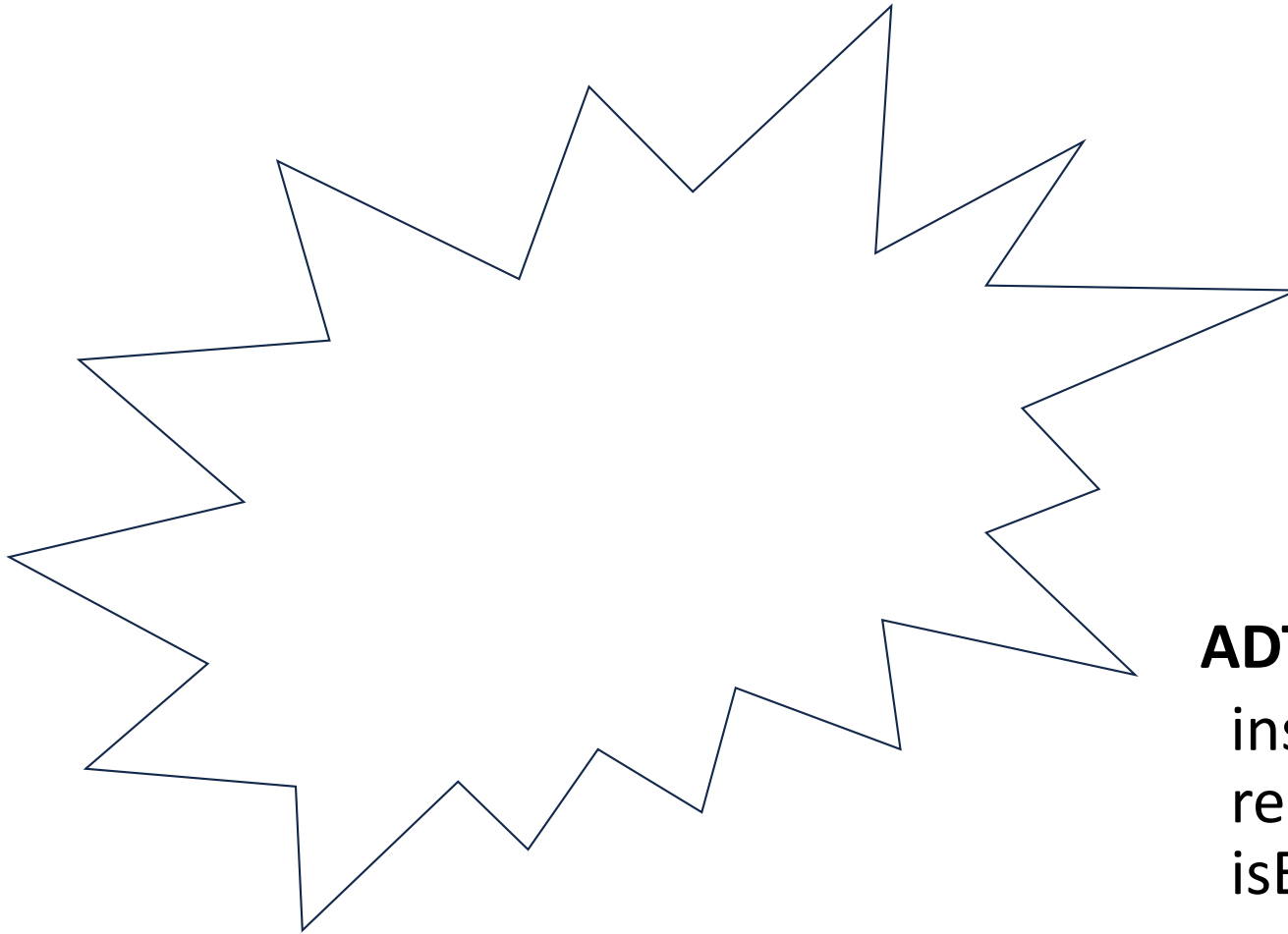
Data Structures

March 27 – Heaps and Priority Queues

Wade Fagen-Ulmschneider, Craig Zilles



Secret, Mystery Data Structure



ADT:

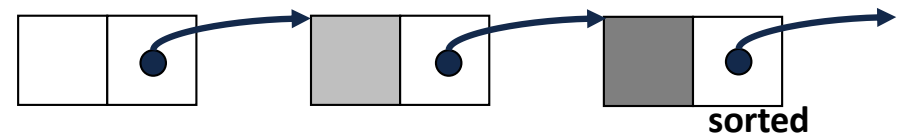
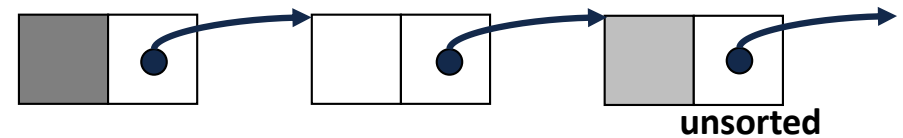
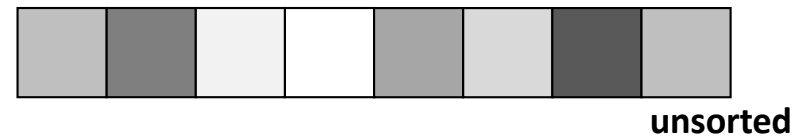
insert

remove

isEmpty

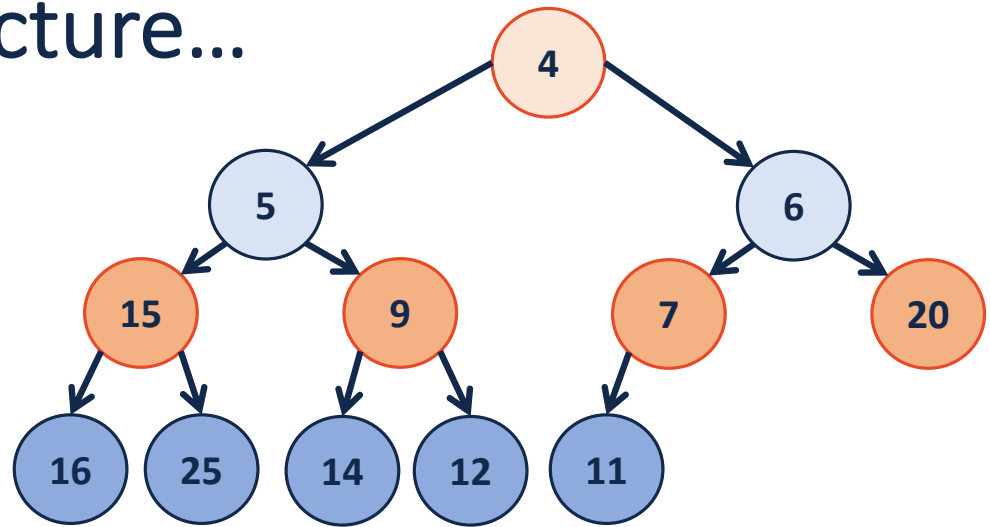
Priority Queue Implementation

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



What all is wrong in this table?

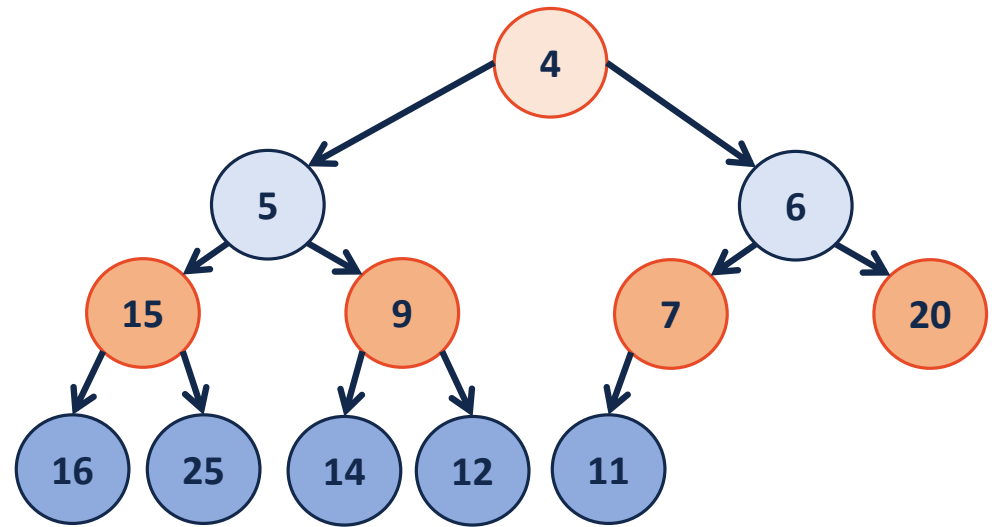
Another possible structure...



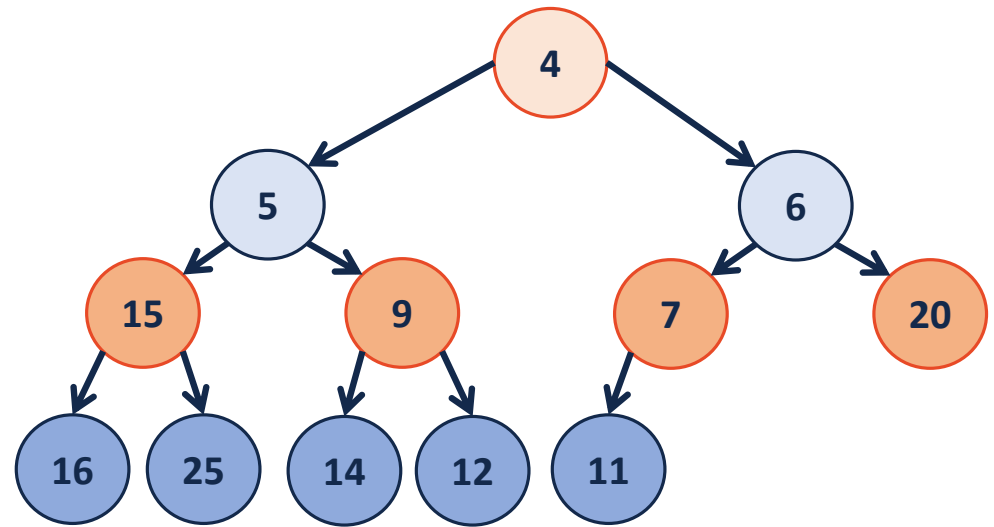
(min)Heap

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			
---	---	---	----	---	---	----	----	----	----	----	----	--	--	--

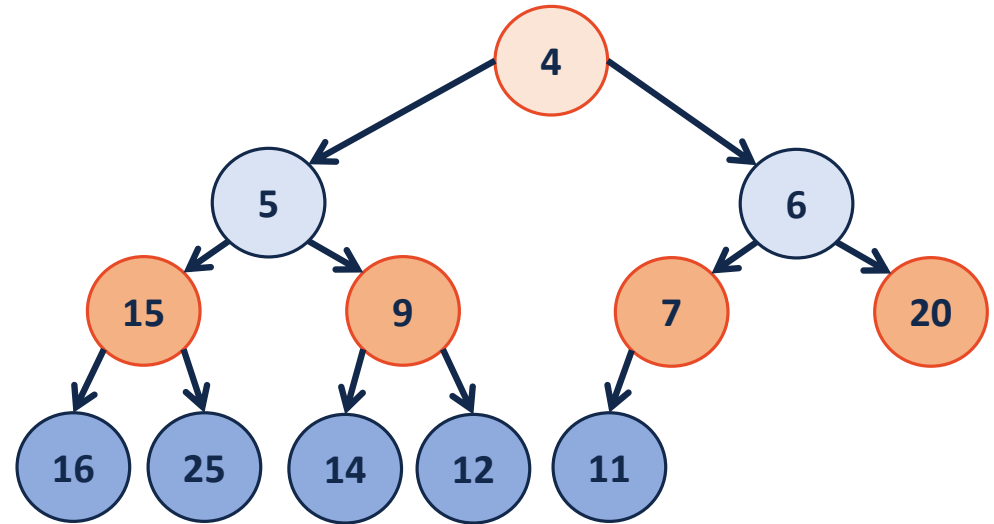


Exam Programming B retrospective

- Some observations:

- Survey incoming:

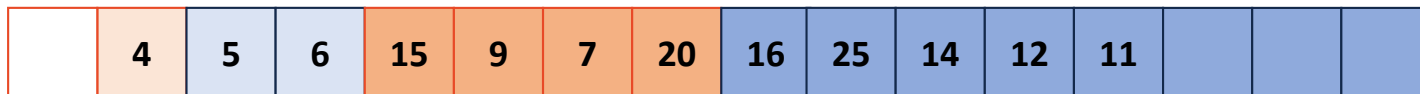
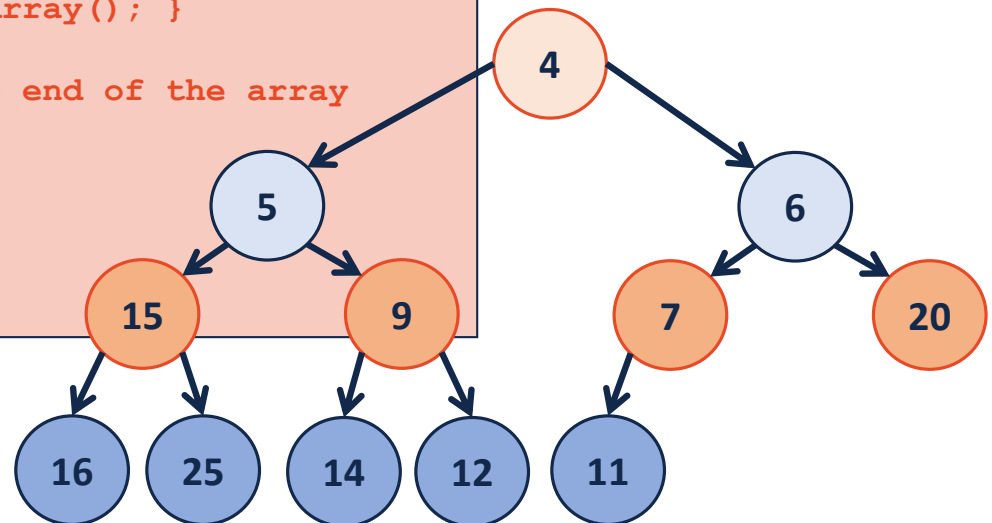
insert



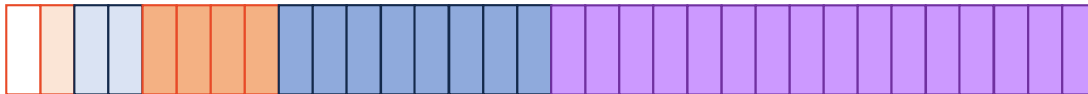
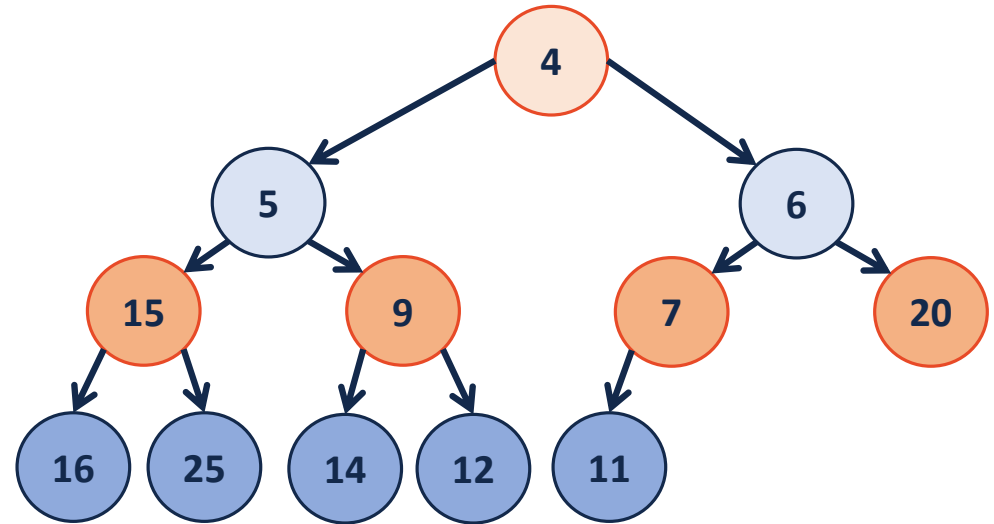
	4	5	6	15	9	7	20	16	25	14	12	11			
--	---	---	---	----	---	---	----	----	----	----	----	----	--	--	--

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



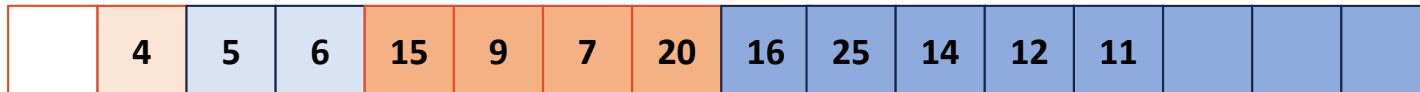
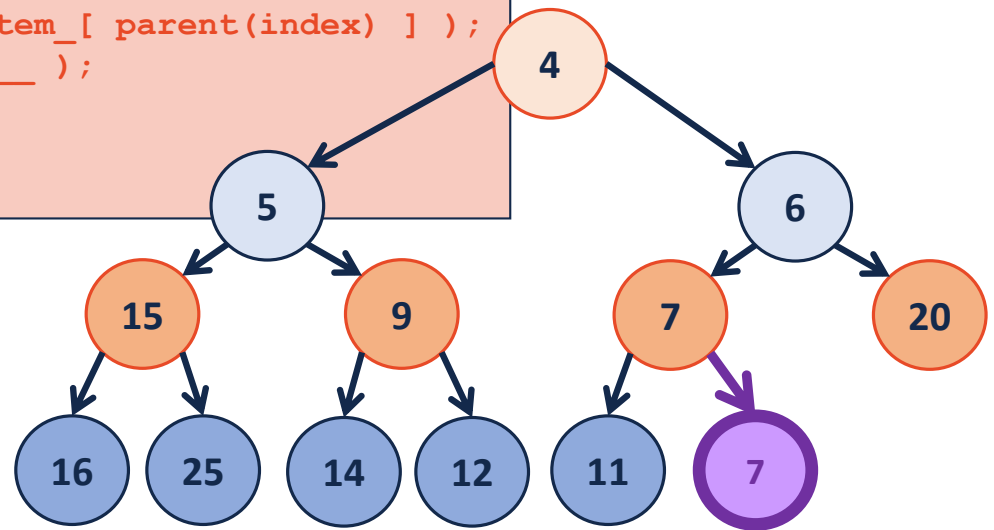
insert - heapifyUp

```
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 }
```

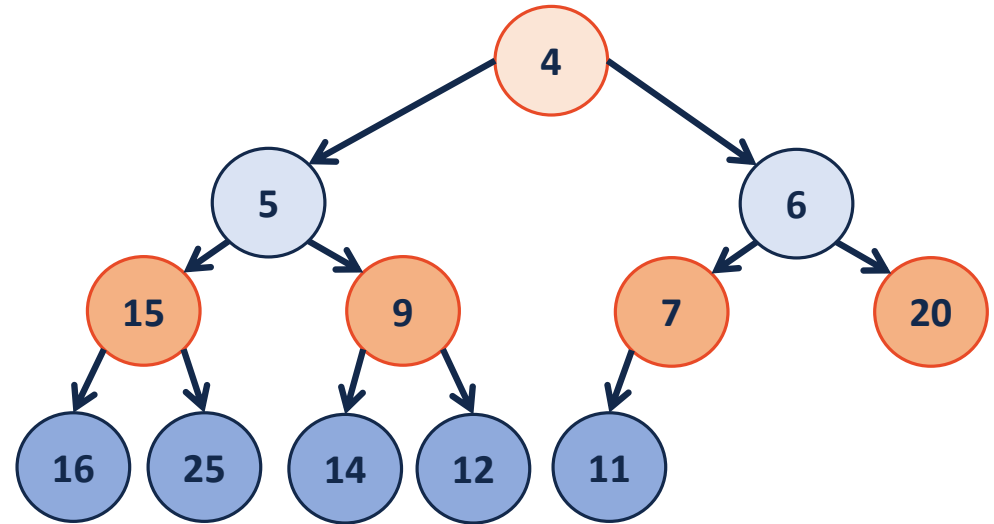
```
1 template <class T>
2 void Heap<T>::_heapifyUp( _____ ) {
3     if ( index > _____ ) {
4         if ( item_[index] < item_[ parent(index) ] ) {
5             std::swap( item_[index], item_[ parent(index) ] );
6             _heapifyUp( _____ );
7         }
8     }
9 }
```

heapifyUp

```
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 }
```



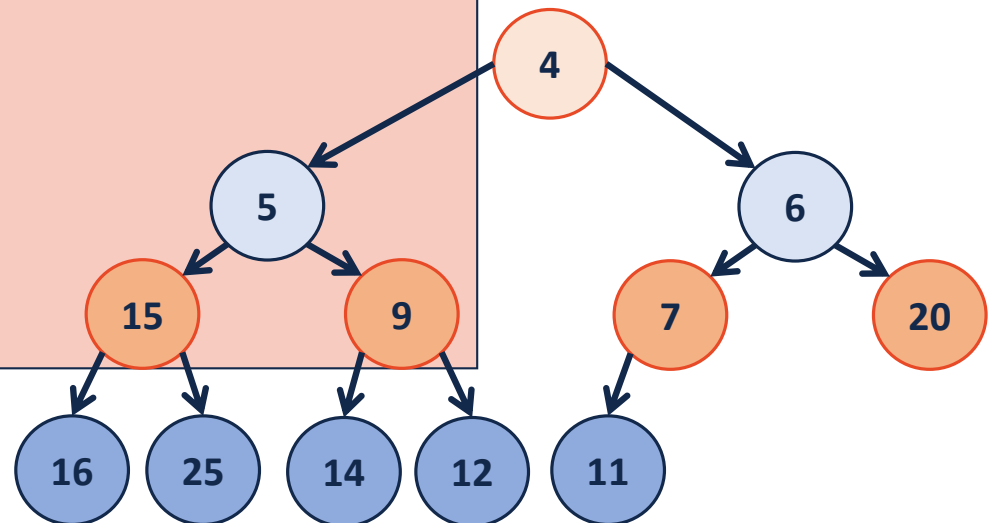
removeMin



	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 }
```

removeMin

```
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 }
```

