STL Data Structures

Tyler Sorey
Invited Lecture
December 3, 2018

Overview

std::array

- Fixed size container
  - Size must be known at compile time
- Realistically, use a vector

```
  // construction uses aggregate
initialization
  std::array<int, 3> arr{ {2, 1, 3} };
  // double-braces required in 11 (not in 14)

  // container operations are supported
  std::sort(arr.begin(), arr.end());

  // ranged for loop is supported
  for(const auto& s: arr)
    std::cout << s << ' ';

  // prints: 1, 2, 3
```

std::vector

- Dynamically sized array
- Contiguous in memory
- Random access: O(1)
- Insertion/Deletion at end: O(1)
- Insertion/Deletion: O(n)
- Size is handled automatically, which is good and bad
  - Allocations are costly as everything is copied on expansion
    - g++ doubles the size each time
- “reserve” is a beautiful thing

```
// Create a vector containing integers
std::vector<int> v;
v.reserve(2);

// Add two more integers to vector
v.push_back(25);
v.push_back(13);

// Iterate and print values of vector
for(int n : v)
  std::cout << n << '\n';

 // prints: 25, 13
```

std::list

- Not contiguous in memory
- Doubly-linked list
- Insertion/Deletion: O(1)*
  - Need iterator to location
  - Fast random access not supported
- Good if splitting/joining lists frequently

```
// Create a list containing integers
std::list<int> l = { 7, 5, 16, 8 };
// Add an integer to the front of the list
l.push_front(25);
// Add an integer to the back of the list
l.push_back(13);
// Insert an integer before 16 by searching
auto it = std::find(l.begin(), l.end(), 16);
if (it != l.end())
    l.insert(it, 42);
// Iterate and print values of the list
for (auto n : l)
    std::cout << n << '\n';

// prints: 25, 7, 5, 42, 16, 8, 13
```

std::deque

- Double-ended queue
  - Fast insertions/deletions at front or back
- Not contiguous in memory
- Elements not copied on storage expansion (cheaper than vector)
- Same big O complexity as vector
- Lots of memory overhead for small deque
  - I.e. for a deque with 1 element it would still allocate it's full internal array, 8 or 16 times the object size libstdc++ or libc++

```
  // Create a deque containing integers
  std::deque<int> d = {7, 5, 16, 8};

  // Add an integer to the beginning and end of
the deque
  d.push_front(13);
  d.push_back(25);

  // Iterate and print values of deque
  for(int n : d)
    std::cout << n << '\n';

  // prints: 13, 7, 5, 16, 8, 25
```

std::map

- Stores key/value pairs
  - All keys must be unique
- Insertion/Deletion/Search are all O(log(n))
- Generally implemented as a Red Black Tree
- Can create a custom comparator for Key sorting

```
template<
class Key,
class T,
class Compare = std::less<Key>,
class Allocator = std::allocator<std::pair<const
Key, T>>
> class map;

auto compare = [](std::string left, std::string
right)
{ return left.length() < right.length(); };

std::map<std::string, std::string,
decltype(compare)> map(compare);
```

std::unordered_map

- Stores key/value pairs
  - All keys must be unique
- Insertion/Deletion/Search are all O(1)
- Generally implemented as a Hash Table
- Can create a custom hashing and key comparison algorithms

```
// Create an unordered_map
std::unordered_map<std::string, std::string> u = {
    {"GREEN","#00FF00"},
    {"BLUE","#0000FF"}
};

// Add new entry to the unordered_map
u["RED"] = "#FF0000";

// Modify an entry
u["GREEN"] = "#NEWVAL";

// Output values by key
std::cout << "The HEX of color GREEN is:[" <<
u["GREEN"] << "]\n";
```

std::multimap/std::unordered_multimap

- Multimaps and unordered_multimaps are very similar to the non multi versions, except they do not require unique keys
- Complexities are the same as their singular counterparts

Container Adaptors

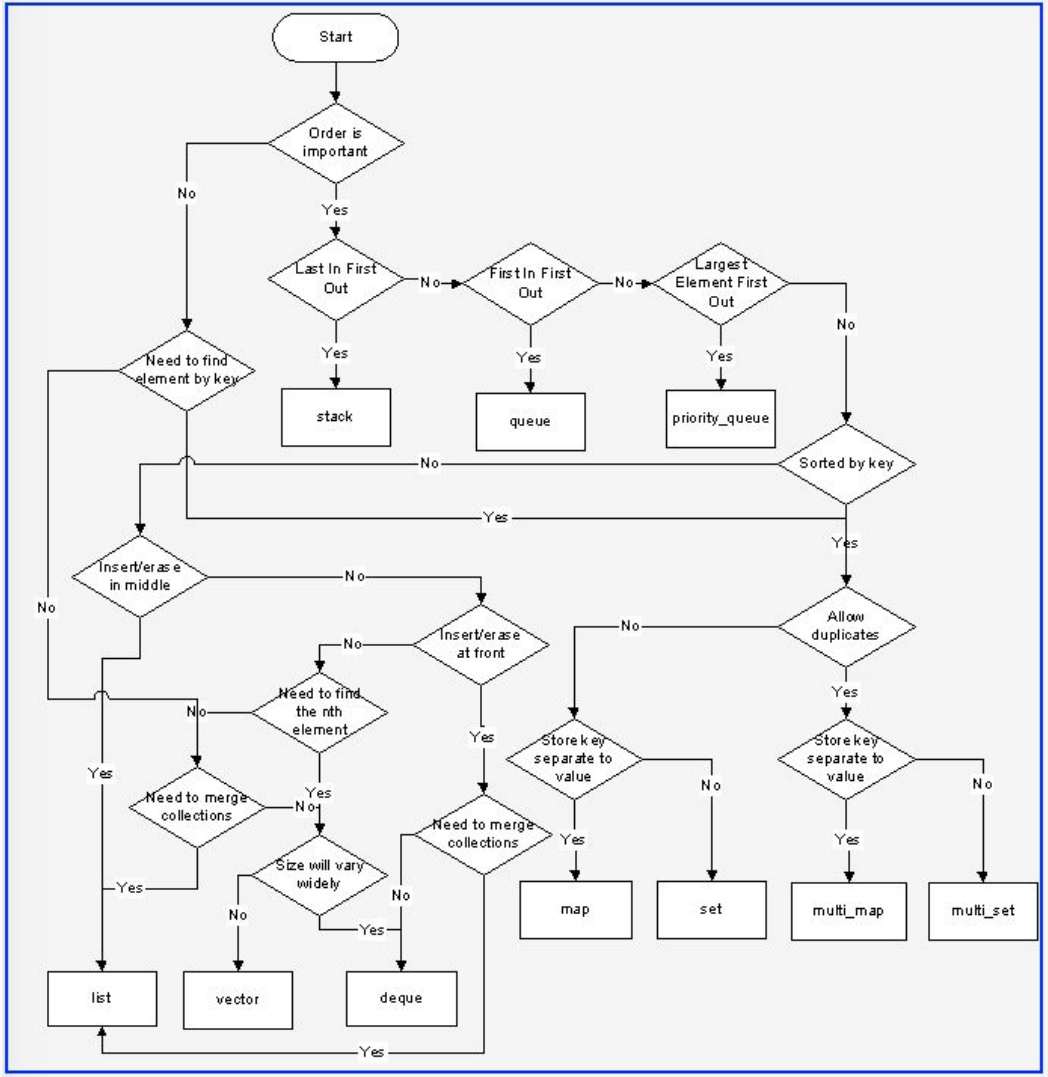
- These act as interfaces to underlying containers

- std::stack
  - LIFO
- std::queue
  - FIFO
- std::priority_queue
  - FIFO with priority

```
template<
  class T,
  class Container = std::vector<T>,
  class Compare = std::less<typename
    Container::value_type>
> class priority_queue;
```

Data Structure Selection

- Most commonly used data structures in my experience: vector & unordered_map



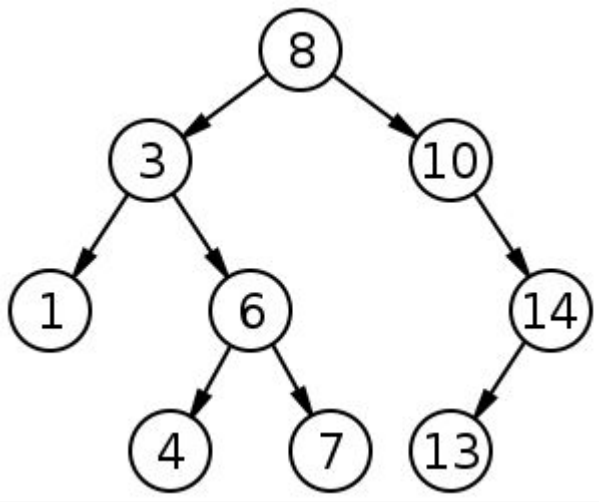
Binary Search Tree vs Balanced Trees

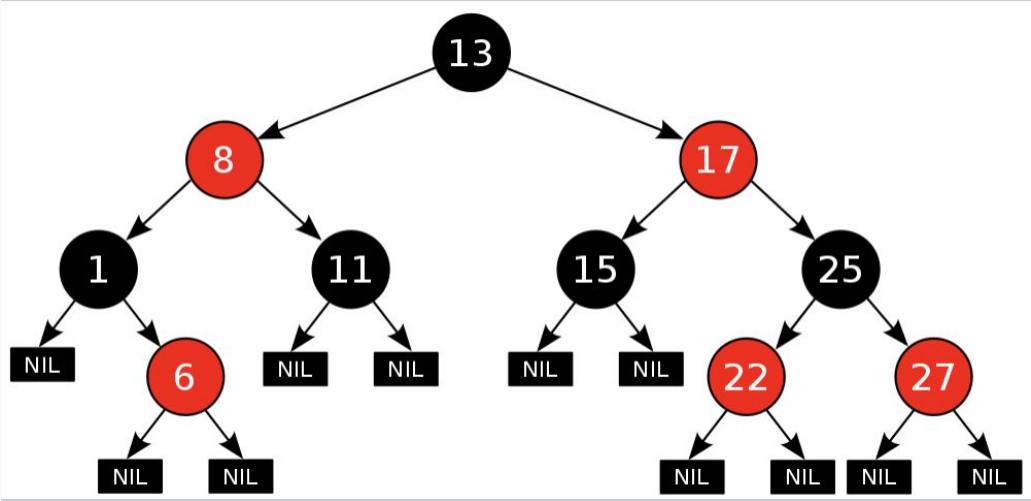
Binary Search Trees

- Worst case search: O(n)

Balanced Trees

- Worst case search: O(log(n))
- Red-Black Tree
- AVL Tree





Examples

```cpp
int main()
{
 bst<int> tree(5);

 tree.addNode(3);
 tree.addNode(7);
 tree.addNode(2);
 tree.addNode(4);

 tree.print();
  return 0;
}
```

```cpp
template <typename T,
 typename = typename std::enable_if_t<std::is_arithmetic<T>::value>>
class bst
{
public:
 bst(T value)
 {
   root = std::make_shared<node<T>>(value);
 }

 auto addNode(T value) -> bool
 {
   return root->addNode(value);
 }

 auto exists(T value) -> bool
 {
   return root->exists();
 }

 auto print() -> void
 {
   root->print();
 }

private:
 std::shared_ptr<node<T>> root;
};
```

BST Example

```
template <typename T,
 typename = typename std::enable_if_t<std::is_arithmetic<T>::value>>
class node
{
public:
 node(T value)
   : val(value)
 {}

 auto addNode(T value) -> bool
 {
   if(getValue() > value)
   {
     if(left == nullptr)
     {
       left = std::make_shared<node<T>>(value);
       return true;
     }
     else
       return left->addNode(value);
   }
   else if(getValue() < value)
   {
     if(right == nullptr)
     {
       right = std::make_shared<node<T>>(value);
       return true;
     }
     else
       return right->addNode(value);
   }
   return false;
 }
```

```
 auto getValue() -> T
 {
   return val;
 }

 auto exists(T value) -> bool
 {
   if(val == value)
     return true;
   else if(getValue() > value)
   {
     if(left != nullptr)
       return left->exists(value);
   }
   else if(getValue() < value)
   {
     if(right != nullptr)
       return right->exists(value);
   }
   return false;
 }

 auto print() -> void
 {
   std::cout << val << std::endl;
   if(left != nullptr)
     left->print();
   if(right != nullptr)
     right->print();
 }

private:
 T val;
 std::shared_ptr<node<T>> left = nullptr;
 std::shared_ptr<node<T>> right = nullptr;
};
```

BST Example

```cpp
template<typename value_type, typename Compare =
std::greater<value_type>>
class heap
{
public:
 auto peek() -> std::optional<int>
 {
   if (container.size() == 0)
     return {};
   return container[0];
 }

 auto pop() -> std::optional<value_type>
 {
   if(container.size() == 0)
     return {};

   auto item = container[0];
   container[0] = container.back();
   container.erase(container.end() - 1);

   fixHeapDown();
   return item;
 }

 auto add(value_type new_item) -> void
 {
   container.push_back(new_item);
   fixHeapUp();
 }

 auto print() -> void
 {
   for(const auto& val : container)
     std::cout << val << std::endl;
 }
```

```cpp
private:
 std::vector<value_type> container;
 Compare compare = Compare();

 auto fixHeapUp() -> void
 {
   int index = container.size() - 1;
   while(hasParent(index) && compare(container[index], getParent(index)))
   {
     swap(getParentIndex(index), index);
     index = getParentIndex(index);
   }
 }

 auto fixHeapDown() -> void
 {
   int index = 0;
   while (hasLeft(index))
   {
     auto swap_child = getSwapChild(index);
     if (compare(container[index], container[swap_child]))
       break;
     else
       swap(index, swap_child);

     index = swap_child;
   }
 }

 auto getSwapChild(int index) -> int
 {
   if(!hasRight(index))
     return getLeftIndex(index);
   if (compare(getRight(index), getLeft(index)))
     return getRightIndex(index);
   return getLeftIndex(index);
 }
```

```cpp
auto getParentIndex(int child_index) -> int
{
  return (child_index - 1) / 2;
}

auto getLeftIndex(int parent_index) -> int
{
  return parent_index * 2 + 1;
}

auto getRightIndex(int parent_index) -> int
{
  return parent_index * 2 + 2;
}

auto hasParent(int index) -> bool
{
  return getParentIndex(index) >= 0;
}

auto hasLeft(int index) -> bool
{
  return getLeftIndex(index) < container.size();
}

auto hasRight(int index) -> bool
{
  return getRightIndex(index) < container.size();
}

auto getParent(int index) -> value_type
{
  return container[getParentIndex(index)];
}

auto getLeft(int index) -> value_type
{
  return container[getLeftIndex(index)];
}
```

Heap Example

```
auto getRight(int index) -> value_type
{
  return container[getRightIndex(index)];
}

 auto swap(int first, int second) -> void
 {
   auto temp = container[first];
   container[first] = container[second];
   container[second] = temp;
 }
};
```

```
int main()
{
 heap<int, std::less<int>> test;

 test.add(15);
 test.add(11);
 test.add(120);
 test.add(1);
 test.add(14);
 test.add(119);

 test.print();
 std::cout << std::endl;

 std::cout << "Peek: " << test.peek().value_or(-1) << std::endl;
 std::cout << std::endl;

 std::cout << "Popped: " << test.pop().value_or(-1) << std::endl;
 std::cout << std::endl;

 test.print();
 std::cout << std::endl;

 std::cout << "Adding 2..." << std::endl;
 test.add(2);

 test.print();

 return 0;
}
```

Heap Example Cont.

```
template <typename key_type, typename value_type>
class hashEntry
{
public:
 hashEntry(key_type new_key, value_type new_value)
 : key(new_key), value(new_value)
 {}

 auto getKey() const -> key_type
 {
   return key;
 }
 auto getValue() const -> value_type
 {
   return value;
 }

 auto setKey(key_type new_key) -> void
 {
   key = new_key;
 }
 auto setValue(value_type new_value) -> void
 {
   value = new_value;
 }

 bool operator==(const hashEntry& other) const
  {
    return key == other.getKey();
  }

private:
 key_type key;
 value_type value;
};
```

```
template <typename key_type, typename value_type>
class hashTable
{
public:
 hashTable()
 {
 }

 auto add(key_type key, value_type value) -> void
 {
   // get the hash value of the key
   auto hash_val = hash(key);
   // look for the key in our list
   auto it = std::find_if(hash_table[hash_val].begin(),
hash_table[hash_val].end(), [key]
     (const hashEntry<key_type, value_type>& other)
     {
       return other.getKey() == key;
     });
   // if this key exists, update the value
   if(it != hash_table[hash_val].end())
     (*it).setValue(value);
   else
     hash_table[hash_val].push_back(hashEntry(key, value));
 }
```

Hash Table Example

```
 auto get(key_type key) -> std::optional<value_type>
 {
   // get the hash value of the key
   auto hash_val = hash(key);

   // look for the key in our list
   auto it = std::find_if(hash_table[hash_val].begin(),
hash_table[hash_val].end(), [key]
     (const hashEntry<key_type, value_type>& other)
     {
       return other.getKey() == key;
     });
   // return it if it exists
   if(it != hash_table[hash_val].end())
     return (*it).getValue();

   return {};
 }

 auto print() -> void
 {
   for(int i = 0; i < TABLE_SIZE; ++i)
   {
     std::cout << "table entry: " << i << std::endl;
     for(const auto& entry : hash_table[i])
     {
       std::cout << "Key: " << entry.getKey() << " - Value: " <<
entry.getValue() << std::endl;
     }
     std::cout << std::endl;
   }
 }
```

```
private:
 std::array<std::list<hashEntry<key_type, value_type>>, TABLE_SIZE>
hash_table;

 auto hash(key_type key) -> int
 {
   auto hash = std::hash<key_type>{} (key);
   return static_cast<int>(hash % TABLE_SIZE);
 }
};
```

Hash Table Example cont.

```
int main()
{
 hashTable<std::string, int> table;

 table.add("1", 10);
 table.add("11", 11);
 table.add("111", 12);
 table.add("1111", 13);
 table.add("11111", 14);
 table.add("111111", 15);
 table.add("1111111", 16);
 table.add("11111111", 17);
 table.add("111111111", 18);

 table.print();

 table.add("111111", 25);

 std::cout << "Value Update Check: " << table.get("111111").value_or(-1)
<< std::endl;

 return 0;
}
```

Hash Table Example cont.

Useful Links

- https://github.com/gibsjose/cpp-cheat-sheet
- https://en.cppreference.com/w/