Lecture 12

SORTED LIST DATA TYPE

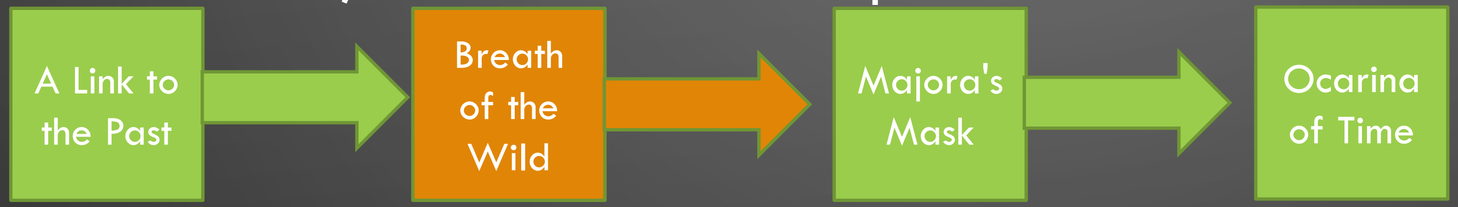
Outline

- What is the Sorted List ADT?
- Creating a Sorted List by adapting the ArrayList ADT
- Creating a Sorted List by adapting the LinkedList ADT
- Tradeoffs and BigO

Sorted List ADT

- Example: Sorted List of Strings, sorted alphabetically

- Unlike a list, insert method finds position on its own: insert("Breath of the Wild")



- Same is possible for removal method: remove("Majora's Mask")

- What other useful methods could we add?

Sorted List ADT

- <u>Overview:</u> A sorted list ADT contains objects, not necessarily distinct but of the same type, sorted by thier value.
- <u>Differences from List:</u>
  - New Methods:
    - remove (sorted)
    - getPosition
  - Changed Method:
    - insert (sorted)
  - Renamed Method:
    - remove:removeAt

```
template <typename T>
class AbstractSortedList
{
  // determine if a list is empty
  virtual bool isEmpty() = 0;

  // return current lenght of the list
  virtual std::size_t getLength() = 0;

  // insert item at ordered position in the list
  virtual void insert(const T& item) = 0;

  // remove first occurance of item from the list
  virtual void remove(const T& item) = 0;

  // remove item at position in the list using 0-based indexing
  virtual void removeAt(std::size_t position) = 0;

  // remove all items from the list
  virtual void clear() = 0;

  // get a copy of the item at position using 0-based indexing
  virtual T getEntry(std::size_t position) = 0;

  // get the position of the first occurance of item or negated position
  // where it would be inserted.
  virtual long int getPosition(const T& newValue) = 0;

};
```

Strategy 1: Adapting Sorted List ADT From ArrayList

- Since Sorted List has so much overlap with List, can we just adapt our ArrayList or LinkedList ADT to make the implementation simpler? Yes!
- Inheritance: A Sorted-List *is-a* List ADT

```
template <typename T>
class ArraySortedList: public AbstractSortedList<T>, private DynamicArrayList<T>
```

- Composition: A Sorted-List *has-a* List

```
template <typename T>
class ArraySortedList: public AbstractSortedList<T>
{

public:
  ...
private:
  DynamicArrayList<T> plist; // private array list
};
```

Strategy 1a: Reusing the ArrayList using Inheritance

Directly overloaded functions:

```
template <typename T>
bool ArraySortedList<T>::isEmpty()
{
  return DynamicArrayList<T>::isEmpty();
}

template <typename T>
std::size_t ArraySortedList<T>::getLength()
{
  return DynamicArrayList<T>::getLength();
}

template <typename T>
void ArraySortedList<T>::removeAt(std::size_t position)
{
  DynamicArrayList<T>::remove(position);
}

template <typename T>
void ArraySortedList<T>::clear()
{
  DynamicArrayList<T>::clear();
}

template <typename T>
T ArraySortedList<T>::getEntry(std::size_t position)
{
  return DynamicArrayList<T>::getEntry(position);
}
```

Adjusted Functions:

```
template <typename T>
void ArraySortedList<T>::insert(const T& item)
{
  // todo
}

template <typename T>
void ArraySortedList<T>::remove(const T& item)
{
  // todo
}

template <typename T>
long int ArraySortedList<T>::getPosition(const T& newValue)
{
  // todo
  return 0;
}
```

Strategy 1b: Reusing the ArrayList using Composition

Directly overloaded functions:

```cpp
template <typename T>
bool ArraySortedList<T>::isEmpty()
{
  return plist.isEmpty();
}

template <typename T>
std::size_t ArraySortedList<T>::getLength()
{
  return plist.getLength();
}

template <typename T>
void ArraySortedList<T>::removeAt(std::size_t position)
{
  plist.remove(position);
}

template <typename T>
void ArraySortedList<T>::clear()
{
  plist.clear();
}

template <typename T>
T ArraySortedList<T>::getEntry(std::size_t position)
{
  return plist.getEntry(position);
}
```

Adjusted Functions:

```cpp
template <typename T>
void ArraySortedList<T>::insert(const T& item)
{
  // todo
}

template <typename T>
void ArraySortedList<T>::remove(const T& item)
{
  // todo
}

template <typename T>
long int ArraySortedList<T>::getPosition(const T& newValue)
{
  // todo
  return 0;
}
```

Implementing the Missing Functions

Insert:

1. Find sorted position in underlying ArrayList object
2. Use ArrayList.insert(position)

Remove:

1. Find sorted position in underlying ArrayList object
2. Use ArrayList.remove(position)

GetPosition:

1. Find sorted position in underlying ArrayList object

Strategy 2a: Reusing the LinkedList using Inheritance

Directly overloaded functions:

```
template <typename T>
bool SortedLinkedList<T>::isEmpty()
{
  return LinkedList<T>::isEmpty();
}

template <typename T>
std::size_t SortedLinkedList<T>::getLength()
{
  return LinkedList<T>::getLength();
}

template <typename T>
void SortedLinkedList<T>::removeAt(std::size_t position)
{
  LinkedList<T>::remove(position);
}

template <typename T>
void SortedLinkedList<T>::clear()
{
  LinkedList<T>::clear();
}

template <typename T>
T SortedLinkedList<T>::getEntry(std::size_t position)
{
  return LinkedList<T>::getEntry(position);
}
```

Adjusted Functions:

```
template <typename T>
void SortedLinkedList<T>::insert(const T& item)
{
  // todo
}

template <typename T>
void SortedLinkedList<T>::remove(const T& item)
{
  if(isEmpty()) throw std::range_error("empty list in remove");

  // todo
}

template <typename T>
long int SortedLinkedList<T>::getPosition(const T& newValue)
{
  // todo
  return 0;
}
```

Strategy 2b: Reusing the LinkedList using Composition

Directly overloaded functions:

```cpp
template <typename T>
bool LinkedList<T>::isEmpty()
{
  return plist.isEmpty();
}

template <typename T>
std::size_t LinkedList<T>::getLength()
{
  return plist.getLength();
}

template <typename T>
void LinkedList<T>::removeAt(std::size_t position)
{
  plist.remove(position);
}

template <typename T>
void LinkedList<T>::clear()
{
  plist.clear();
}

template <typename T>
T LinkedList<T>::getEntry(std::size_t position)
{
  return plist.getEntry(position);
}
```

Adjusted Functions:

```cpp
template <typename T>
void SortedLinkedList<T>::insert(const T& item)
{
  // todo
}

template <typename T>
void SortedLinkedList<T>::remove(const T& item)
{
  if(isEmpty()) throw std::range_error("empty list in remove");

  // todo
}

template <typename T>
long int SortedLinkedList<T>::getPosition(const T& newValue)
{
  // todo
  return 0;
}
```

BigO Tradeoffs

- Why back a SortedList with an ArrayList vs a LinkedList?
- Which, if either, implementation has better algorithmic order (and what is it) for:

	ArrayList	LinkedList
RemoveAt(position)		
GetPosition(value)		
GetEntry(position)		
Remove(value)		
Insert(value)		

Assignment/Homework

- P3 due this Thursday
- ICE 6 will be released this afternoon