Lecture 13: Array Data Storage

Introduction

An array is a fixed-size collection and a fundamental storage mechanism for data structures. In linear (sequential) data structures, arrays provide efficient random access, validation, and capacity management. Certain operations for special scenarios, such as insertion and removal, can have constant runtime complexity. The array storage is partitioned into the *dataset* (actual content of the data structure) and free space, which is usually managed using a single index reference.

ArrayList Interface

The *ArrayList* interface provides various methods for managing array-based storage:

- `append(itm)` - adds *itm* to the end of the collection.
- `insert(idx,itm)` - inserts *itm* at index *idx* in the collection.
- `remove(itm)` - removes the first occurrence of *itm* from the collection.
- `detach()` - removes the last item from the collection.
- `find(itm)` - returns the index of the first occurrence of *itm* in the collection.
- `contains(itm)` - checks if *itm* exists in the collection.
- `at(idx)` - returns the item at index *idx* in the collection. (equivalent to operator[])
- `empty()` - checks if the collection is empty.
- `length()` - returns the size of the collection.

These methods are commonly used for managing dynamic arrays and ensuring efficient data storage. Their implementations will use a single index reference and an array named *size* and *data*, respectively.

Capacity Methods

The capacity methods, `empty()` and `length()`, operate in constant time $O(1)$ because they directly reference the *size*, which tracks the dataset's length.

`empty()`
1. return *size* is 0.

`length()`
1. return *size*.

Access Methods

Since arrays allow random access, the `at()` method operates in constant time $O(1)$. However, `contains()` and `find()` require linear traversal $O(n)$.

`contains(itm)`
1. $i \leftarrow 0$.
2. while $i < size$ and $data[i] \neq itm$ do
   1. $i \leftarrow i + 1$.
3. return $i \neq size$.

`find(itm)`
1. $i \leftarrow 0$.
2. while $i < size$ and $data[i] \neq itm$ do
   1. $i \leftarrow i + 1$.
3. return i.

`at(idx)`
1. if *idx* in $[0, size)$, then
   1. return $data[idx]$.
2. throw an out-of-bound error.

Insertion Methods

To insert into a dataset, existing elements may need to be shifted to maintain order.

Example (Inserting f at index 2):
Initial dataset: [a,b,c,d,e]

Operation	Dataset
shift $4 \to 5$	[a,b,c,d,e,e]
shift $3 \to 4$	[a,b,c,d,d,e]
shift $2 \to 3$	[a,b,c,c,d,e]
insert f	[a,b,f,c,d,e]

The actual `insert()` method algorithm is

`insert(idx,itm)`
1. if $idx \leq size$ and $size < data$`.length`, then
   1. $i \leftarrow size$
   2. while $i > idx$ do
      1. $data[i] \leftarrow data[i\text{-}1]$.
      2. $i \leftarrow i$ - 1.
   3. $data[idx] \leftarrow itm$.
   4. $size \leftarrow size + 1$.

The `append()` method avoids shifting, resulting in $O(1)$ runtime:

`append(itm)`
1. if $size < data$`.length`, then
   1. $data[size] \leftarrow itm$.
   2. $size \leftarrow size + 1$.

Removal Methods

To remove an element, values must be shifted left to fill the gap and *size* is decremented.

Example (Removing index 2):
Initial dataset: [a,b,c,d,e]

Operation	Dataset
shift $3 \to 2$	[a,b,d,d,e]
shift $4 \to 3$	[a,b,d,e,e]

Finally, *size* is decremented to prevent access to the removed element.

The `remove()` method, in addition, initially locates the item before performing the remove but maintains a linear time $O(n)$:

`remove(itm)`
1. if $size > 0$, then
   1. $i \leftarrow 0$
   2. while $i < size$ and $data[i] \neq itm$ do
      1. $i \leftarrow i + 1$.
2. if $i \neq size$, then
   1. $size \leftarrow size$ - 1.
   2. while $i < size$ do
      1. $data[i{+}1] \leftarrow data[i]$.
      2. $i \leftarrow i + 1$.

The `detach()` method removes the last element in constant time $O(1)$:

`detach()`

1. if *size* > 0, then
   1. *size* $\leftarrow$ *size* - 1.

Resizing Functions

For dynamic arrays, a resize() method is needed to increase capacity. The process involves:

1. Copies its content to a temporary array.
2. Reallocates (deallocate then allocate) new memory to expand its size.
3. Copies back its content from the temporary array.

An algorithm of a `resize()` method that doubles its size is

`resize()`

1. $t \leftarrow$ new T[*size*].
2. $i \leftarrow 0$.
3. while $i <$ *size* do
   1. $t[i] \leftarrow$ *data*$[i]$.
   2. $i \leftarrow i + 1$.
4. deallocate *data*.
5. *data* $\leftarrow$ new T[2**size*].
6. $i \leftarrow 0$.
7. while $i <$ *size* do
   1. *data*$[i] \leftarrow t[i]$.
   2. $i \leftarrow i + 1$.
8. *size* $\leftarrow$ *size* * 2.

This method operates in $O(n)$ time complexity due to data copying.