TCSS 342 – Data Structures

Abstraction

Abstraction

- What is abstraction?
  - The abstract refers to the world of ideas.
  - This is in contrast to the physical world which we call the concrete.
- Examples:
  - A blueprint is an abstraction and the building is the concrete.
  - A battle plan is an abstraction and the battle is the concrete.
  - Democracy is an abstraction and the US Government is a concrete.
  - An algorithm is an abstraction and the program is a concrete.
- Are the abstractions identical to the concretes?
  - No, they are idealizations.
  - Some abstractions are never made concrete (i.e. unicorn).

Abstraction in Computer Science

- Abstraction is the process of eliminating details so as to arise at a concise conception or model of some entity, or process.
  - Abstraction is applied in computer science when determining the problem to solve, the resources to use, and the algorithms to employ.
- Concretization is the process of filling in details of a model or concept so as to arrive at a real world product of the abstraction.
  - Concretization (implementation) is applied in computer science when we have to transform an abstract algorithm into a program.

Abstract Data Types

- An abstract data type (ADT) is an idea of how to organize information.
  - One such idea is to organize it in a list.
- An abstract data type has two dimensions:
  - It's abstract definition in the minds of programmers.
  - It's concrete realization in lines of program code.
- We use what is called an application programing interface (API) to help us translate between these two dimensions.
  - The API of an ADT helps us understand how to use it from the outside.
  - The API of an ADT is a specification for implementation from the inside.
  - As long as the implementers obey the API then the user of the ADT can be confident it will work according to the API.
  - This is the foundation of libraries and shared applications.

Abstract Data Types

- Here is a list of abstract data types we'll cover in this course and some of their implementations:
  - List:
    - Array, Array List, Linked List, Stack, Queue
  - Tree:
    - Binary Heap, Binary Search Tree, AVL Tree, Red-Black Tree, B-Tree, Huffman Tree
  - Map/Dictionary:
    - Hash tables
    - Hash functions
    - Chaining, Probing
    - Extendible Hashing, Linear Hashing
  - Graphs:
    - Undirected, directed, weighted, connected, bipartite and complete graphs.

TCSS 342 – Data Structures

Lists

Invariant

- An invariant of an abstract data type is a property (or set of properties) that does not change after performing operations.
  - Typical operations are addition, deletion and searching for elements.
  - It is important to maintain the invariant so that these operations can be reliably performed again.
- In Java many data structures are organized as collections.
  - A collection is a very abstract data type that has many different implementations:
    - Stack, Queue, Array, Linked List, Red-Black Tree, Hash Table
- Collection Invariant: A group of objects (elements) stored together.
  - At the very least this means that additions, deletions and searches should not upset the other elements.
- Example: Deleting an interior element of a linked list.
  - If we merely delete the pointer to the element we risk losing all of the remaining elements.
  - Instead we reassign the pointer to the next element to maintain the invariant.

List

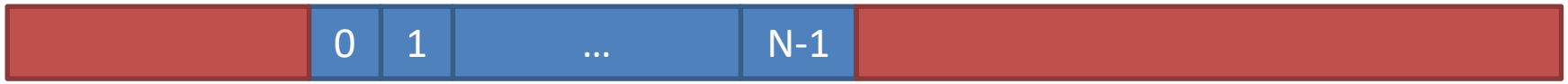
- List Invariant: A list is a collection of objects in *some* order.
  - The order of the elements should also be maintained, but which order depends on the kind of list.
- A list can be further defined by its access criteria through additional invariants:
  - Stack Invariant: first-in last-out
  - Queue Invariant: first-in first-out
  - Array (Vector) Invariant : index access
  - Linked List Invariant : iterator access

Arrays

- What does this line do?

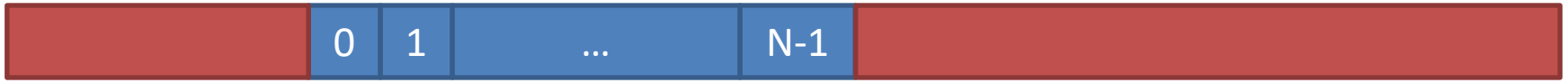
```
int[] ints = new int[size];
```

- It allocates enough memory in sequence to store `size` integers. How many bytes is that?
  - Each integer takes 32 bits or 4 bytes in modern systems.
  - So in total we will have `size` * 4 bytes.



- This segment of memory is then used to store the entire array of integers.
  - The address of the array is the address of the first element.
  - We can calculate the address of any element in the array with some simple math.
    - Address of array + 4 * index of element.
  - This is called random access.
  - The primary benefit of random access is that it is constant time access.

Arrays



- What if we want to add more elements to our array? Can we extend this memory region?
  - The adjacent region is likely occupied by other program data.
  - These static arrays that cannot grow in size.
  - If we need more space we must:
    1. Allocate a new larger region.
    2. Copy the contents of the array over.
  - Given n elements in an array growing the array will take O(n) operations. So we don't want to grow the array too often!
- If we want an array that grows in this way we must use an ArrayList in Java.
  - This is an implementation of a dynamic array.

Dynamic Arrays

- Dynamic arrays like Java's ArrayList will begin with an initial capacity.
  - This capacity can initially be set to just 1 but this is uncommon.
  - Typically this is a small power of 2 like $2^4 = 16$.
- The dynamic array keeps track of both the capacity and the size of the array.
  - Elements can be added simply when the size is less than the capacity. This takes $O(1)$ or constant time.
  - When the size reaches capacity we must grow the array.
- Growing the array requires a few steps.
  - First we declare a new array that has twice the capacity.
  - Then we copy all the elements from the old array to the new array.
  - In total this takes $O(n)$ or linear time.

Amortized Constant Time

- The worst case runtime of adding to a dynamic array is $O(n)$ however most of the adds will actually take $O(1)$.
  - We use an analytical technique called amortization to spread the cost of doubling over the other adds.
- Consider adding N elements to a dynamic list where N is large.

- Every time we double it costs us the current size of the array in operations but this also means we double the number of adds we make before the next doubling.
  - Can we find out how many operations it would cost to add N elements to an empty dynamic array?

Amortized Constant Time

- Say $N = 100$ and we start with an array of capacity one.
  - Cost of 100 adds is 100.
  - Cost of doubling will be first 1 then 2 and so on
$$1 + 2 + 4 + 8 + 16 + 32 + 64 = 127$$
  - So the total cost is 227 a little more than twice the size of the array.
- Say $N = 1024$ and we start with an array of capacity one.
  - Cost of 1024 adds is1024.
  - Cost of doubling will be first 1 then 2 and so on
$$\sum_{i=0}^{9} 2^i = 1023$$
  - So the total cost is 2047 almost exactly twice the size of the array.
- Can we prove this in general?

Amortized Constant Time

- From our examples we can calculate the cost of N adds will be

$$N + \sum_{i=0}^{\lceil \log N \rceil - 1} 2^i$$

- What is the simplified form of this geometric series?

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

- So

$$N + \sum_{i=0}^{\lceil \log N \rceil - 1} 2^i = N + 2^{\lceil \log N \rceil - 1 + 1} - 1$$

$$\leq N + 2^{\log N + 1} - 1$$

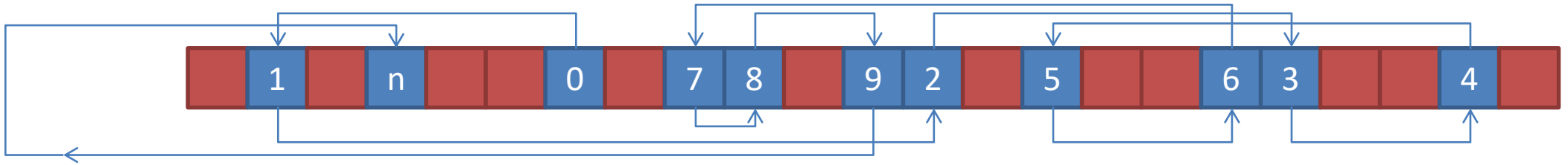
$$= N + 2N - 1$$

$$= 3N - 1$$

- While in the worst case a single add might take $O(n)$ operations if we do many adds we will probably not notice this cost since adding n elements also only takes $O(n)$.
  - This is called the amortized cost because we spread the cost of the expensive operation over many operations.

Linked Lists

- A linked list is a structure of nodes where each node is linked to others by storing the addresses of other nodes.
  - This allows for dynamic growth because every time an element is added a new node is allocated somewhere in memory and linked to the rest.
  - So the worst case cost of adding to the end of a linked list is $O(1)$.



- What are the drawbacks of the linked list model?
  - We have given up random access.
  - If we want item i we must traverse the links to get there. This will take $O(i)$ time.

Arrays and Linked Structures

- Every data structure we study in this course is implemented with basic data structures.
  - Arrays
  - Recursively linked structures or classes
- The framework of linked structures is a recursive reference.
  - We terminate the recursion with a null reference.
  - This simple structure is called a linked list.

```
private class Node {
        Type item;
        Node next;

        Node(Type it, Node n) {
                item = it;
                next = n;
        }

        public String toString() {
                return item.toString();
        }
}
```

TCSS 342 – Data Structures

Bubble Sort

Searching a List

- Finding an element in a list is a standard problem.
  - If the list is unsorted how do we find the element?
    - Brute force: We check every location.
    - This is called linear search because of running time is $O(n)$.
  - If the list is sorted how do we find the element?
    - Binary search.
    - This search procedure is $O(\log n)$.
- So searching a sorted list is easier (faster) than searching an unsorted list.
  - This means we might have reason to keep our lists in order.
- How hard is it to sort an unsorted list?
  - This is the problem of sorting:

    Sort a List (SORT)
    Input: $A[1 \ldots n]$ is a list of integers
    Output: A' a permutation of A such that $A'[1] \leq A'[2] \leq \ldots \leq A'[n]$.
  - There are many solutions to this problem.

Bubble Sort

```
private static void bubbleSort(List<Integer> list){
   for(int i = 0; i < list.size()-1; i++){
      boolean inversion = false;
         for(int j = 0; j < list.size()-i-1; j++){
            // check for inversion between item j and j+1
            // if so swap them
            if(list.get(j)>list.get(j+1)){
               inversion = true;
               swap(list,j,j+1);
            }
         }
      if(!inversion)
         break;
      }
}
```

- How many operations does it take this method to sort the input list?
  - This depends on the size of the list so we will say the list is of size n.
  - We can probably use our intuition to guess. What is your guess?
    - $O(n^2)$
  - We want to be able to build our intuition by improving our formal analysis skills.

Operations

- What counts as an operation?
  - Adding two numbers together?
  - Multiplying two numbers?
  - Comparing two strings to see if they are the same?
- What counts as an operation depends on the resolution of our analysis.
  - At a high-level of analysis we usually count arithmetic between two numbers as a single operation.
  - However, at a finer level of detail we might want to count every bit operation that occurs during those additions and multiplications.
- When we are analysing Java code we usually operate on a higher level and count all the following operations as taking constant time.
  - Declarations
  - Assignments
  - Arithmetic
  - Logic
  - Branching
  - Many method calls (check the APIs of your data structures).

Constant Time

- We say constant time as a way of saying that an operation takes $O(1)$ time to complete.
- Recall:
  - $O(1)$ is the set of constant functions.
  - We use 1 to represent this set but constant functions can be any constant.
  - Examples: $1, 42, 1337, 1701,$ 1 billion
  - When a constant is unknown we give them names.
    - Usually we use lowercase early letters like:

      $$a, b, \boldsymbol{c}, d, e$$

    - You can take any constant and put it through any function and get another constant:

      $$a^2, 5b, c^d \cdot e!$$

- Example: The ArrayList methods *get(index)* and *set(index, value)* are listed on the Oracle documentation as taking constant time.
  - We don't know which constants these are but we can just call it c_{get} and c_{set}.
- Swapping two ArrayList elements takes two *get*s and two *set*s plus another c_e extra work.
  - This means the whole swap method takes:

    $$2(c_{get} + c_{set}) + c_e$$

  - This is still a constant so we can just call it c_{swap}.

Bubble Sort

```
private static void bubbleSort(List<Integer> list){
   for(int i = 0; i < list.size()-1; i++){
      boolean inversion = false;
         for(int j = 0; j < list.size()-i-1; j++){
            // check for inversion between item j and j+1
            // if so swap them
            if(list.get(j)>list.get(j+1)){
               inversion = true;
               swap(list,j,j+1);
            }
         }
      if(!inversion)
         break;
      }
}
```

- How many operations does it take bubble sort to sort an ArrayList of size n?
  - All of the ArrayList method calls used above are constant time.
    - Is this true if we use a LinkedList instead?
  - So the bulk of the work in this method is carried out by the nested loops.
- We can express the run time of a loop with a summation.

Bubble Sort

```
private static void bubbleSort(List<Integer> list){
   for(int i = 0; i < list.size()-1; i++){
      boolean inversion = false;
         for(int j = 0; j < list.size()-i-1; j++){
            // check for inversion between item j and j+1
            // if so swap them
            if(list.get(j)>list.get(j+1)){
               inversion = true;
               swap(list,j,j+1);
            }
         }
      if(!inversion)
         break;
      }
}
```

$$f(n) = \sum_{i=0}^{n-1} \left[a + \sum_{j=0}^{n-i-1} b \right]$$

- This is a nested sum because we have a nested loop.
- The inner sum represents the cost of the inner loop.
- It is important that we use different indexes on our sums just like it is important to use different indexes in our code.

Summation Simplification

$$f(n) = \sum_{i=0}^{n-1} \left[a + \sum_{j=0}^{n-i-1} b \right]$$

$$= \sum_{i=0}^{n-1} a + \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-1} b$$

$$= a \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} b \sum_{j=0}^{n-i-1} 1$$

$$= an + \sum_{i=0}^{n-1} b \sum_{j=0}^{n-i-1} 1$$

$$= an + \sum_{i=0}^{n-1} b(n-i)$$

$$= an + b \sum_{i=0}^{n-1} (n-i)$$

Summation Simplification

$$f(n) = an + b\sum_{i=0}^{n-1}(n-i)$$

- There are two ways to handle this sum. One straightforward and one clever.
- First the straightforward method.

$$= an + bn\sum_{i=0}^{n-1} 1 - b\sum_{i=0}^{n-1} i$$

$$= an + bn^2 - b\sum_{i=0}^{n-1} i$$

- We still need to know what this sum results in:

$$\sum_{i=0}^{n-1} i$$

- This is an arithmetic series. I usually call it Gauss' Sum.

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Summation Simplification

$$f(n) = an + bn^2 - \frac{bn(n-1)}{2}$$
$$= an + bn^2 - \frac{bn^2}{2} + \frac{bn}{2}$$
$$= an + \frac{bn^2}{2} + \frac{bn}{2}$$

Now for the clever method:

$$f(n) = an + b\sum_{i=0}^{n-1}(n-i)$$
$$= an + b\sum_{i=1}^{n} i$$
$$= an + \frac{bn(n+1)}{2}$$
$$= an + \frac{bn^2}{2} + \frac{bn}{2}$$

Both ways we arrive at the same result. We can conclude our big-oh bound:

$$f(n) \in O(n^2)$$

TCSS 342 – Data Structures

Insertion Sort

Insertion Sort

```
private static void insertionSort(List<Integer> list) {
   // insert the ith item into items 0 to i-1 in the proper position
   for(int i = 1; i < list.size(); i++){
      for(int j = i; j > 0 && list.get(j) < list.get(j-1); j--){
         swap(list,j,j-1);
      }
   }
}
```

- Can we express this method's worst case running time as a summation?

$$f(n) = \sum_{i=1}^{n-1} \left[a + \sum_{j=1}^{i} b \right]$$

- Can we simplify the summation now that we have it?

Summation Simplification

$$f(n) = \sum_{i=1}^{n-1} \left[a + \sum_{j=1}^{i} b \right]$$

$$= a(n-1) + b \sum_{i=1}^{n-1} i$$

$$= a(n-1) + \frac{bn(n-1)}{2}$$

$$= \frac{bn^2}{2} + \left(a - \frac{b}{2} \right) n - a$$

$$f(n) \in O(n^2)$$

- So Insertion Sort, like Bubble Sort, is also $O(n^2)$.

TCSS 342 – Data Structures

Selection Sort

Selection Sort

```
private static void selectionSort(List<Integer> list){
   // search for the ith smallest item
   // and put it in the ith position
   for(int i = 0; i < list.size() - 1; i++){
      int index = i;
      for(int j = i + 1; j < list.size(); j++){
         if(list.get(j) < list.get(index)){
            index = j;
         }
      }
      swap(list, i, index);
   }
}
```

- Can we express this method's worst case running time as a summation?

$$f(n) = \sum_{i=0}^{n-2} \left[a + \sum_{j=i+1}^{n-1} b \right]$$

- Can we simplify the summation now that we have it?

Summation Simplification

$$f(n) = \sum_{i=0}^{n-2} \left[a + \sum_{j=i+1}^{n-1} b \right]$$

$$= a(n-1) + \sum_{i=1}^{n-2} b \sum_{j=i+1}^{n-1} 1$$

$$= a(n-1) + \sum_{i=1}^{n-1} b \left((n-1) - (i+1) + 1 \right)$$

$$= a(n-1) + \sum_{i=1}^{n-1} b \, (n-i-1)$$

$$= an - a + \frac{bn^2}{2} - \frac{bn}{2}$$

$$f(n) \in O(n^2)$$

- So Selection Sort, like Bubble Sort and Insertion Sort, is also $O(n^2)$.