CS 261-020
Data Structures

Lecture 3
C Basics
Dynamic Array vs. Linked List
1/23/24, Tuesday
Odds and Ends

• All office hours posted – at KEC 1130

• Recitation 2: instead of +3, you are allowed to make up for full credits by the beginning of your next week’s recitation time

• Recitation 3 posted
  • You will have more time working on recitation 2 😊
Odds and Ends

• Sign up for Demos:
  • Where? – TA Hours page on Canvas
  • Assignment 1
    • Code due: Sunday (Jan 28)
    • demo due: Friday of week 5 (Feb 9)
  • For those who already signed up for demos this week, you are allowed to reschedule or demo the assignment right away.
    • No 2nd demo after the due date, i.e., only one demo allowed per assignment
Hints for recitation 2
Lecture Topics:

• C Basics
  • Function pointer
• Dynamic Array
• Linked List
Recap: C Basics – Void Pointers (void*)

• A void pointer is a pointer represented by the type `void*`.
• A void pointer is a generic pointer, it can point to data of any data type.
  • E.g., a void pointer points to an integer
    ```
    int var = 20;
    void *v_ptr = &var;
    ```
• Why `void*`?
  • It allows the data structures to contain data of any type while remaining type agnostic
The null character is important → indicates **the end of the string**

Functions rely on ‘\0’:
- `printf()` – know when to stop processing the string
- `strlen()` – returns the number of characters in a string
  - Count until it finds a null character

Allocating memory to store a string: num of char + null char
- Q: How many char can we store in the `str`?
  ```
  char* str = malloc(64 * sizeof(char));
  63
  +1 for '\0'
  ```
• Constant strings in C:
  `char* name = "Harry Potter";`
  `char name[64]`

• Constant strings are read-only, thus cannot be modified.
  `name[0] = 'l'; //illegal but no error message`

• Best to mark it be constant
  `const char* name = "Harry Potter";`
  `name[0] = 'l'; //illegal with compiling error`
C Basics – strings in C (cont.)

• Useful functions for C strings: \#include <string.h>
  • `strlen()` – returns the number of characters in the string
  • `strncpy()` – copy a specified number of characters from one string to another
  • `snprintf()` – “printing” content into a string, up to a specified number of characters
    • From `<stdio.h>`
    • `strcmp()` – compare two strings, returns 0 if they are equal
      ```c
      if(strcmp(str1, str2) == 0)
      ```
  • And many more… check `string.h`
Function pointers allows us to store the memory address of a function in a variable and use that memory address to call the function being pointed to.

- Allows us to pass functions as arguments to other functions

Why would we want to pass a function as an argument to another function?

Consider this...
C Basics – Function pointers

• Write a function to sort an array
• To make it work for any type,
  • each element is void*, and thus the pointer to the array is void**

```
void sort(void** arr, int n); //can pass arr of any type into this function
```

• Question: How does sort() be able to compare the values in the array?
  • Use function pointers!
  • The function calling our sort() and passing data into it does know these things
C Basics – Function pointers

• Add a function pointer:
  • A function that compares two values from the array to be sorted and return a value indicating which is bigger/smaller
  • `int cmp(void* a, void* b);`

  • So our `sort()` becomes to:
  `void sort(void** arr, int n, int (*cmp)(void* a, void* b));`
C Basics – Function pointers

• To use this function pointer
  • the calling function will need access to a function for *comparing* elements, i.e., integers
  • This function will have to match the prototype of the function pointer argument to our sort()
  • E.g.,
    ```c
    int compare_ints(void* a, void* b) {
        int* ai = a, *bi = b; /* Cast void* back to int*. */
        if (*ai < *bi)
            return 0;
        else
            return 1;
    }
    ```
  • Function call will be:
    ```c
    sort((void**)array_of_ints, number_of_ints, compare_ints);
    ```
C Basics – Function pointers

```c
void sort(void** arr, int n, int (*cmp)(void* a, void* b));
```

• Within sort():
  • Whenever we need to compare two values from the array being sorted, we can just call cmp()

```c
if (cmp(arr[i], arr[j]) == 0) {
    /* Put arr[i] before arr[j] in the sorted array. */
}
else {
    /* Put arr[i] after arr[j] in the sorted array. */
}
```

• Demo....
Lecture Topics:

• C Basics
• Dynamic Array
• Linked List
Abstract Data Type (ADT)

• Abstract Data Type (ADT) – a mathematical model for data types
• Specifies:
  • the type of data stored
  • the operations supported on them
  • the types of parameters of the operations.

• Why “abstract”? 
  • an implementation-independent view of the data type
Dynamic Arrays

• Elements in an array are stored in a contiguous block of memory
• Allow random access (direct access)
  • i.e., time to access the 1st element = time to access the last element
  • By using array subscript ([i]):
    ```c
    int* array = malloc(1000 * sizeof(int));
    array[0] = 0;
    array[999] = 0;
    ```

• Demo...
Dynamic Arrays (cont.)

• Basic operations:
  • get – Gets the value of the element stored at a given index in the array
  • set – Sets/updates the value of the element stored at a given index in the array
  • insert – Inserts a new value into the array at a given index.
    • Sometimes, dynamic array implementations limit insertion to a specific location in the array, e.g. only at the end.
  • remove – Removes an element at a given index from the array
    • Sometimes, dynamic array implementations avoid moving elements up a spot by only allowing the last element to be removed
Dynamic Arrays (cont.)

• Drawbacks:
  • Fixed size, must be specified when the array is created
    • For static array:
      ```c
      int array[50];
      ```
    • For dynamic array:
      ```c
      int *array = malloc (50 * sizeof(int));
      ```
  
  → Need to allocate more memory if we need to store more data
    • How?

• Dynamic array DS doesn’t have a fixed capacity
  • Has a variable size and can grow as needed
Dynamic Arrays (cont.)

• Need to keep track of three things:
  • **data** – underlying data storage array
  • **size** – number of elements currently stored in the array
  • **capacity** – number of elements data has space for before it must be resized

• How it works?
  • An array of known capacity is maintained by the dynamic array DS.
  • As elements are inserted, they are simply stored in **data**
  • If an element is inserted into the dynamic array, and there isn’t capacity for it in the underlying data storage array (**data**), the **capacity of the underlying data storage array is doubled**. Then the new element is inserted into this larger data storage array.
### Dynamic Arrays

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>8</th>
<th>1</th>
<th>4</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>
Inserting an element into dynarray

• Case 1: if size < capacity
  • At least one free spot in data
  • Insert the new element

• Case 2: if size == capacity
  • No free spot in data
  • Step 1: allocate a new array that has twice the capacity
  • Step 2: copy all elements from data to new array
  • Step 3: delete the old data array
  • Step 4: Insert the new element
Another Example

• Insert 16 to the following dynamic array:

  data
  size = 4
  capacity = 4

  5
  13
  8
  31

  new_data

• Step 1: allocate a new array that has twice the capacity
Another Example

• Insert 16 to the following dynamic array:

```
data
size = 4
capacity = 4
```

```
5
13
8
31
```

```
new_data
5
13
8
31
```

• Step 2: copy all elements from data to new array
Another Example

• Insert 16 to the following dynamic array:

• Step 3: delete the old data array and update data
Another Example

• Insert 16 to the following dynamic array:

```
data
size = 5
capacity = 8
```

![Dynamic Array Diagram]

• Step 4: Insert the new element
Lecture Topics:

• C Basics
• Dynamic Array
• Linked List
Linked List

- **Linear Data Structure**
- Elements in a linked list are stored in **nodes** and chained together
  - Not in contiguous memory
  - Thus, no random access
- A linked list in which each node points only to the next link in the list is known as a singly-linked list.
  - E.g.:

```c
struct node {
    void* val;
    struct node* next;
};
```
Linked List

• Always contains as many nodes as it has stored values
  • Add an element $\rightarrow$ allocate a node, add it to the list
  • Remove an element $\rightarrow$ free the node from the list

• Many forms of linked list:
  • Keeps track only of the first element in the list, known as head
Linked List

• Many forms of linked list:
  • Keeps track only of the first element in the list, known as head
  • Keeps track of both the head of the list and the tail, or last element
Linked List

• Many forms of linked list:
  • Keeps track only of the first element in the list, known as head
  • Keeps track of both the head of the list and the tail, or last element
  • Each node keeps track of both the next link and the previous link in the list, known as a doubly-linked list
Linked List

• Many forms of linked list:
  • Keeps track only of the first element in the list, known as head
  • Keeps track of both the head of the list and the tail, or last element
  • Each node keeps track of both the next link and the previous link in the list, known as a doubly-linked list
  • Last node points to the first node, known as circular-linked list
Linked List

• Many forms of linked list:
  • With sentinels, which are special nodes to designate the front/end of the list
    • E.g.: a doubly-linked list using both front and back sentinels
Inserting an element into linked list

• Where can we insert?
  • Front/head
  • End/tail
  • Middle
Inserting an element into linked list

• Insert an element to the front:
  • Construct a node to be inserted, new_node
  • Initialize new_node’s next to NULL

• Case 1:
  • Head is NULL (the list is empty)
  • Simply let head point to new_node

• Case 2:
  • Head is not NULL (the list is not empty)
  • new_node’s next points to the 1st node (head);
  • head point to new_node
Inserting an element into linked list

• Insert an element to the end:
  • Construct a node to be inserted, new_node
  • Initialize new_node’s next to NULL

• Case 1:
  • Head is NULL (the list is empty)
  • Simply let head point to new_node

• Case 2:
  • Head is not NULL (the list is not empty)
  • Loop to find the last element, last_node
  • last_node’s next points to the new_node;
Inserting an element into linked list

• Insert an element to the middle:
  • Construct a node to be inserted, new_node
  • Initialize new_node’s next to NULL

• Case 1:
  • Head is NULL (the list is empty)
  • Simply let head point to new_node

• Case 2:
  • Head is not NULL (the list is not empty)
  • Loop to find the position to insert, after_this_node
  • new_node’s next points to the after_this_node’s next
  • after_this_node’s next points to the new_node
Removing and element from a linked list

• Opposite steps as inserting a new one
• Ex. Assuming the list is not empty, and we want to remove the node containing the value 8:
Removing and element from a linked list

• Step 1:
  • Loop to find the node to be removed and the node before, i.e., current points to before_node, node_to_remove points to the node to be removed
Removing and element from a linked list

- Step 2:
  - Set current’s next to node_to_remove’s next
Removing and element from a linked list

• Step 3:
  • free node_to_remove
Next Lecture:

• Complexity Analysis
  • Big O