Data Structures

Rob Bocchino
NASA Jet Propulsion Laboratory
October 22, 2024

Jet Propulsion Laboratory
California Institute of Technology

Introduction

- A **data structure** is
  - A collection of data
  - An interface and implementation for operating on the collection
- Picking good data structures is key to writing code that
  - Is correct and performs well
  - Is understandable and maintainable
- This section of the course will
  - Review some basic data structures and their operations
  - Sketch the implementations
  - Provide some suggestions for picking data structures

Data Structures in FSW

- Many standard data structures allocate and free elements on demand
- In FSW we do not do that
- FSW data structures must
  - Use a fixed total size of memory to hold their data
  - Call *malloc* or *new* only at FSW startup
- We will focus on data structures that meet these requirements

Data Structure Operations

- **Insert:** Add an element to a collection
- **Remove:** Take an element out of a collection
- **Find:** Search for and return an element
- **Iterate:** Visit each element

Basic Data Structures

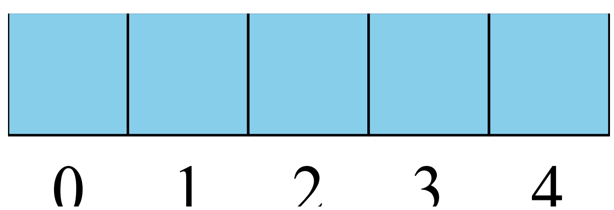
- **Array:** An indexed collection of fixed size
- **List:** An ordered, non-indexed collection
- **Set:** An unordered collection
- **Map:** An unordered mapping from keys to values
- **Queue:** A collection that supports insert and remove only

Implementing Data Structures

- Building blocks
  - C/C++ arrays to hold data elements
  - Pointers or array indices for links between elements
- For FSW, prefer arrays to pointers
  - Often, you don't need a linked data structure at all: just use an array
  - If you do need links, then
    - All memory is pre-allocated at FSW startup
    - So all memory has an associated index into a pre-allocated array
    - Using the index is safer, because it can be bounds-checked
- **Avoid C++ STL** (it uses hidden *new* and *delete* operations)
- Consider using ETL: https://www.etlcpp.com

Array

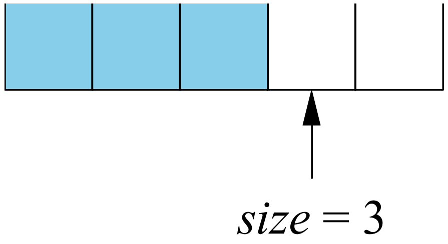
- Data representation: A C/C++ array
- Operations
  - **Insert:** N/A
  - **Remove:** N/A
  - **Find:** Bounds-checked access to the underlying array
  - **Iterate:** Loop over elements



Array: Recommended Use

- Use if
  - The number of elements is fixed; and
  - The elements map to indices; and
  - The index set is numeric and dense
- Don’t use if
  - The number of elements grows or shrinks; or
  - There is no numeric index set; or
  - There is a sparse numeric index set

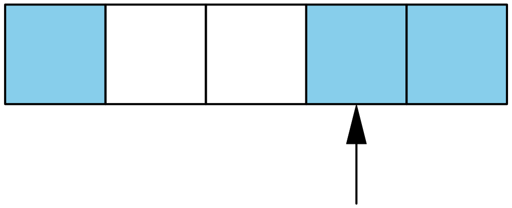
Array-Based LIFO Queue (Stack)



- LIFO means “last in, first out”
- Data representation
  - An array *A* of elements
  - A variable *size* that stores the data size. Initially it is zero.
- Operations (check that *size* is in bounds)
  - **Insert (enqueue):** Add the element at *A[size]* and increment *size*
  - **Remove (dequeue):** Decrement *size*
  - **Find:** N/A
  - **Iterate:** N/A
- Recommended use: If you need LIFO behavior

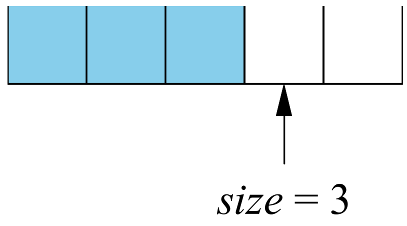
Array-Based FIFO Queue

- FIFO means "first in, first out"
- Implementation is similar to LIFO, but it
  - Uses a circular array (index mod array size)
  - Tracks starting position and size
  - Performs enqueue at one end of the array, dequeue at the other
    - Dequeue moves *start* forward
- Exercise:
  - Sketch the implementation
  - Pay special attention to the cases where the queue is full and empty
- Example: *Utils/Types/CircularBuffer.{hpp,cpp}* in F Prime



$start = 3$, $size = 3$

Array-Based Set/Map: Data Representation



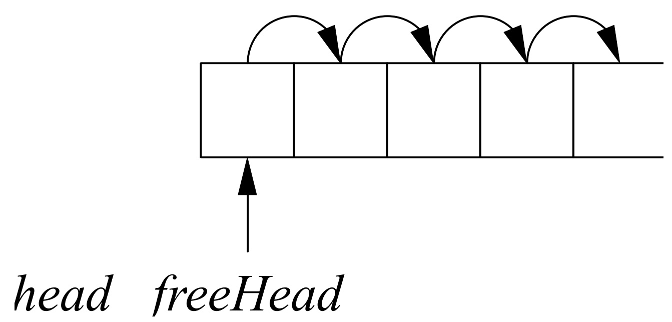
- Data representation
  - An array *A* of
    - Elements (small elements); or
    - Indices into elements stored in a different array (large elements).
  - A variable *size* that stores the current size. Initially it is zero.
- Operations (check that *size* is in bounds)
  - **Insert:** Add the element at *A[size]* and increment *size*
  - **Remove:**
    - Swap the element with the element at *A[size - 1]*
    - Decrement *size*
  - **Find:** Linear search of first *size* elements of *A*
  - **Iterate:** Loop over the first *size* elements of *A*

Array-Based Set/Map: Recommended Use

- Use if
  - The find operation is rare; or
  - The number of elements is small
- Don’t use if
  - Find is common and the number of elements is large
  - Use hash set/map instead
- Variant: Mark nodes as unused instead of swapping nodes
  - Avoids moving nodes around
  - But insert is slower (have to search for an unused node)
  - Example: Command Dispatcher component in F Prime

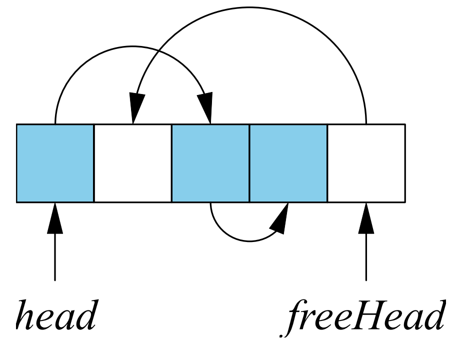
Linked List: Data Representation

- An array *A* of nodes
  - Each node has a member *next* that stores a link to the next node or a special *NONE* value that is distinct from any index.
- A variable *head* that stores the index of the first node in the list
  - Initially it contains *NONE*.
- A variable *freeHead* that stores the index of the first node in the free list
  - Initially all nodes in *A* are in the free list:
    - *freeHead = 0*
    - for *0 <= i < n – 1 do A[i].next = i + 1*
    - *A[n - 1].next = NONE*



Linked List: Operations

- **Insert:**
  - *tmp = A[freeHead].next*
  - *A[freeHead].next = head*
  - *head = freeHead*
  - *freeHead = tmp*



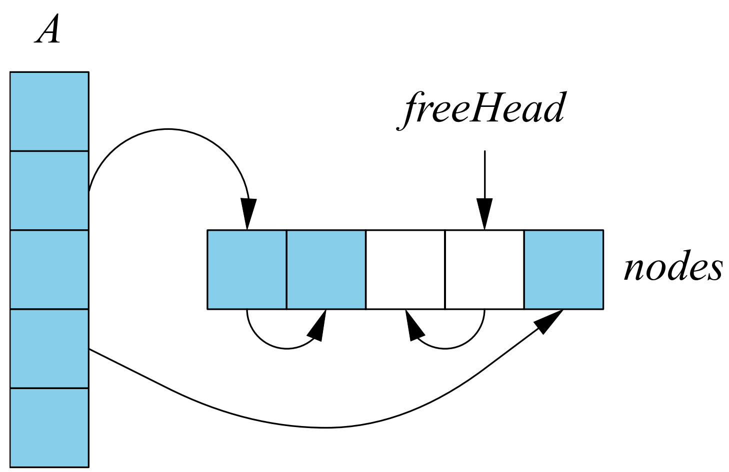
- **Remove:**
  - Let *N* be the node to remove. Search through the linked nodes for *N*, starting at *head*. Maintain a reference *L* to the previous link.
  - Set *L.next = N.next* in general, *head* = *N.next* at the front
  - Insert *N* into the free list.
- **Find:** Linear search through the linked nodes, starting at *head*
- **Iterate:** Loop over the linked nodes, starting at *head*

Linked List: Recommended Use

- Use this if you want several lists to share nodes from a common array
- Otherwise use an array-based set/map
  - It is significantly less complex

Hash Set/Map: Data Representation

- An array *A* of linked lists that share the same array of nodes
- Each list node stores a key only (set) or a key-value pair (map)
- Initially the lists are all empty



Hash Set/Map: Operations

- **Insert:**
  - Use a hash function to convert the key into an index *I* into *A*
  - Insert the key into the linked list at *A[I]*
- **Remove:**
  - Use the hash function to convert the key into an index *I*
  - Perform a remove on the list at *A[I]*
- **Find:**
  - Use the hash function to convert the key into an index *I*
  - Perform a find on the list at *A[I]*
- **Iterate:** Iterate over the array and each list in the array

Hash Set/Map: Recommended Use

- Use if
  - There is no numeric index set for the data; or
  - The index set is sparse; or
  - The data set is large, and you need fast find capability
- Otherwise use an array or array-based set/map
- Notes
  - You need to choose a good hash function, and test the performance
  - Interleaving inserts and finds can cause nondeterministic performance
  - If you do all inserts at initialization time, this is not a problem
- Example: *TlmChan* component in F Prime

References

- Lewis and Denenberg, *Data Structures & Their Algorithms*.
- Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*.