ADTS
2012 10 16

ADTs

- Positional container
  - Write(index, value)
  - Read(index)
- Container ordered on key
  - Write(key, value)
  - Read(key)
- First In First Out - Fifo or Queue
  - Write(value) or Enqueue(value)
  - Read or dequeue
  - First arrived in queue is first served
- Last in first out - Lifo or Stack
  - Write(value) or push()
  - Read or pop()
  - Last arrived is served

ADTs

- Hash table
  - Write(key, value)
  - Read(key)

Common points to adt

- Unbounded
- Implemented with linked list or unbounded array (only unbounded for hash table)

Queue

- write
- Read
- First arrived is first served (FIFO. First in first out)

QUEUE

- Implemented with array (bounded)
- Implemented with resizable array (unbounded)
- Implemented with positional array
- Implemented with linked list

Queue

- Implemented with positional container
- Queue
  - Queue(value) → positionalcontainer.writeInLastPosition(value)==
  - positionalcontainer.write(last, value)
- Dequeue
  - Dequeue() → positionalcontainer.read(0)

Queue – complex/time

Operation	Positional container, implem w linked list	Positional container implem w bounded array	Positional container w unbounded array	Circular array, bounded
Queue (write(last))	(assuming tail pointer), O(1) or constant	O(1)	O(1) normally O(n) if full	O(1)
Dequeue (read(0)	O(1)	O(n elements) (elemnts are shifted)	O(n elements) (elemenst are shifted) And also O(n) when not full enough	O(1)

Pos container w Unbounded array

- Implementation: initial array size S, n =number of elements
  - Write: if there is space, normal write on array
  - If full, malloc a larger (+50%) array a1
  - copy old array a on a1
  - free (a)
- Write(index, value)
  - Constant Omega(1)
  - O(n) when full
  - Average? Constant + 1/S *k * S

- Implementation: initial array size S, n =number of elements
  - Read : normal read on array, constant
  - If not full enough (n/S < 0.5)
  - malloc a smaller (-50%) array a1
  - copy old array a on a1
  - free (a)
  - Read(index)
    - Constant Omega(1)
    - O(n) when not full enough
    - Average? Constant + 1/S *k * S

Queue – complex/space

	Positional container, implem w linked list	Positional container implem w bounded array	Positional container w unbounded array
N = number of elements S = size of array	Descriptor = 2 pointers, one integer = 2*4 + 1* 4 = 12bytes = constant C1 Nodes: N * size of node = N * (size of element + pointer) = C1 + N * size of node	Descriptor = C2 S * size of element	
Pointer – 4 bytes Integer – 4 bytes		If N << S wastes memory	

'full' array scenario

- Linked list
- N= 1M
- Element = 4 bytes
- Pointer = 4 bytes
- Total memory

N * (4 + 4) = 8 Mbytes

- Bounded array
- N = 1 M
- Element = 4bytes
- S = 1.2M
- Total memory

S * 4bytes = 4.8 Mbytes

‘empty’ array scenario

- Linked list
- N= 1K
- Element = 4 bytes
- Pointer = 4 bytes
- Total memory

N * (4 + 4) = 8 Kbytes

- Bounded array
- N = 1 k
- Element = 4bytes
- S = 1.2M
- Total memory

S * 4bytes = 4.8 Mbytes

Complexity analysis

- On time
- On space
- Usually impossible to have best performance both on time and space, trade off is needed (see unbounded array, unbounded is superior in time if memory is wasted)
- No best choice in all cases, depends on scenario (see memory occupation and full/empty array case)

Stack

- Write or push
- Read or pop

  - Implementation

STACK

- Implemented with array (bounded)
- Implemented with resizable array (unbounded)
- Implemented with positional array
- Implemented with linked list

Stack with positional container

- Push()
  - PositionalContainer.write(afterlast, value)
  - Afterlast is integer == number of elements in positional container
  - PositionalContainer.writeAfterLast(value)
- Pop
  - positionalContainer.read(last)
  - Last is integer == number of elements -1
  - PositionalContainer.readLast()

Stack– complex/time

Operation	Positional container, implem w linked list	Positional container implem w bounded array	Positional container w unbounded array	array, bounded
Push (write(last))	Assuming we have pointer to tail O(1)	O(1) No elements to shift (cfr queue)	O(1) if not full O(n) when full	O(1)
Pop (read(last-1))	O(1)	O(1)	O(1) O(n) when reduce size because not enough full	O(1)

Stack – complex/space

	Positional container, implem w linked list	Positional container implem w bounded array	Positional container w unbounded array
N = number of elements S = size of array	Descriptor = 2 pointers, one integer = 2*4 + 1* 4 = 12bytes = constant C1 Nodes: N * size of node = N * (size of element + pointer) = C1 + N * size of node	Descriptor = C2 S * size of element	
Pointer – 4 bytes Integer – 4 bytes		If N << S wastes memory	

Positional container– complex/time

Operation	Positional container, implem w linked list	Positional container implem w bounded array	Positional container w unbounded array
Write (shifts elements) Two parts, finding element and adding	Adding is constant, finding no If pointer to tail Worst case, write before last: O(n-1) Best case, write(0): Average case: O(n/2)	Finding is constant, adding no Worst case: write(0), O(n) Best case: write(last), O(1) Average case: write(middle) O(n/2)	If not full same as bounded If full, write part is constant, copy part is O(n)
Read (cancels element)	Same as write	Same as write	If occupation index (n/S) ok, same as bounded Else same as write

Positional– complex/space

	Positional container, implem w linked list	Positional container implem w bounded array	Positional container w unbounded array
N = number of elements S = size of array	Descriptor = 2 pointers, one integer = 2*4 + 1* 4 = 12bytes = constant C1 Nodes: N * size of node = N * (size of element + pointer) = C1 + N * size of node	Descriptor = C2 S * size of element	
Pointer – 4 bytes Integer – 4 bytes		If N << S wastes memory	

Ordered container– complex/time

Operation	Ordered container, implem w linked list	Ordered container implem w bounded array	Ordered container w unbounded array
Write(key) 2 parts, find right position, insert	Find part worst (n-1), best (1), average O(n/2) Insert part: konst Binary search not useful because requires going through all elements anyway	Find part If no binary search Same as linked list Insert part Average n/2 If binary search ln2(n) Insert n/2	
Read(key) No cancels	Find part only Same as above	If no binary search n/2 If binary search ln2(n)	