CS213/293 Data Structure and Algorithms 2025

Lecture 3: स्टैक और क्यू (Stack and queue)

Instructor: Ashutosh Gupta

IITB India

Compile date: August 9, 2025

Topic 3.1

स्टैक (Stack)

स्टैक (Stack)

Definition 3.1

एक स्टैक एक कंटेनर होता है जहां एलिमेंट्स को आखरी अंदर और पहले बाहर (LIFO) क्रम के अनुसार जोड़ा और हटाया जाता है। (A stack is a container where elements are added and deleted according to the last-in,first-out (LIFO) order.)

- जोड़ने को pushing कहा जाता है। (Addition is called pushing)
- हटाने को popping कहा जाता है। (Deleting is called popping)

Example 3.1

- कॉपियर में कागजों का ढेर (A stack of papers in a copier)
- एडिटर्स में अनडू-रीडू सुविधाएं (Undo-redo features in editors)
- ब्राउज़र पर वापस बटन (Back button on the Browser)

Reference: https://en.cppreference.com/w/cpp/container/stack

स्टैक इंटरफ़ेस में चार फ़ंक्शन होते हैं। (Stack supports four interface methods.)

- `stack<T> s` : नए स्टैक *s* को आवंटित करता है। (Allocates a new stack s.)
- `s.push(e)` : एलिमेंट *e* को स्टैक के शीर्ष पर डालता है। (Pushes the given element e to the top of the stack.)
- `s.pop()` : स्टैक से शीर्ष एलिमेंट को हटाता है। (Removes the top element from the stack.)
- `s.top()` : स्टैक के शीर्ष एलिमेंट को वापस करता है। (Accesses the top element of the stack.)

कुछ सहायक फ़ंक्शन (Some support functions)

- `s.empty()` : स्टैक खाली है या नहीं, यह जांचता है (checks whether the stack is empty)
- `s.size()` : एलिमेंट्स की संख्या वापस करता है (returns the number of elements)

Exercise 3.1
हम स्टैक क्यों परिभाषित करते हैं जब हम उसी काम के लिए वेक्टर का उपयोग कर सकते हैं?
(Why do we define stack when we can use vector for the same effect?)

Commentary: Answer: vector in C++ promises to provide efficient random access, but stack does not make such a promise. Therefore, the implementations of the stack may make implementation choices that may result in inefficient random access.

स्टैक के axioms* (Axioms of stack*)

मान लें s1 और s स्टैक हैं। (Let s1 and s be stacks.)

- `Assume(s1 == s); s.push(e); s.pop(); Assert(s1==s);`
- `s.push(e); Assert(s.top()==e);`

`Assume(s1 == s)` का मतलब है कि हम मानते हैं कि s1 और s के एलिमेंट्स समान है।
(`Assume(s1 == s)` means that we assume that the contents of s1 and s are the same.)

`Assert(s1 == s)` का अर्थ है कि हम जांचते हैं कि s1 और s के एलिमेंट्स समान है।
(`Assert(s1 == s)` means that we check that the contents of s1 and s are the same.)

Exercise: खाली स्टैक पर कार्रवाई (action on the empty stack)

Exercise 3.2

मान लें s एक खाली स्टैक है। (Let s be an empty stack in C++.)

- जब हम `s.top()` कॉल करेगें हैं तो क्या होता है? (What happens when we run `s.top()`?)
- जब हम `s.pop()` कॉल करेगें हैं तो क्या होता है? (What happens when we run `s.pop()`?)

ChatGPT से पूछें। (Ask ChatGPT.)

Commentary: Answer: `s.top()` will cause a segmentation fault. `s.pop()` will not cause any error and exit without any effect. Actually, the behavior depends on the compiler. Compare the behavior of the program under g++ 13.3.0 and 15.1.1.

Topic 3.2

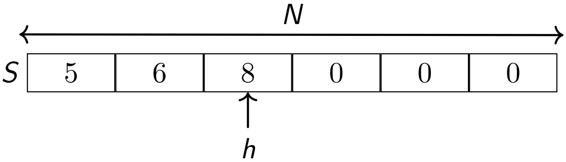
स्टैक के लिए कोड (Implementing stack)

एरे-आधारित स्टैक (Array-based stack)

हम एक सरलीकृत ऐरे-आधारित कोड पर नजर डालते हैं जो एक इंटीजर्स का स्टैक हो।
(Let us look at a simplified array-based implementation of an stack of integers.)

स्टैक में तीन वेरिएबल्स होते हैं। (The stack consists of three variables.)

- N स्टैक में वर्तमान में उपलब्ध स्थान को जगह को बताता है। (N specifies the currently available space in the stack.)
- S एक इंटीजर ऐरे है जिसका आकार N है। (S is the integer array of size N)
- h स्टैक के हेड का स्थान है। (h is the position of the head of the stack.)



स्टैक के लिए कोड (Implementing stack)

```
class arrayStack {
  int  N = 2;      // आकार (Capacity)
  int* S = NULL;   // ऐरे का पॉइंटर (pointer to array)
  int  h = -1;     // स्टैक का वर्तमान हेड (Current head of the stack)
public:
  arrayStack() { S = (int*)malloc(sizeof(int)*N ); }
  int  size()  { return h+1;  }
  bool empty() { return h<0;  }
  int  top()   { return S[h]; } // खाली स्टैक पर क्या होगा? (On empty stack what happens?)
  void push(int e) {
    if( size() == N ) expand(); // स्टैक के आकार को बढ़ाएं (Expand capacity of the stack)
    S[++h] = e;
  }
  void pop() { if( !empty() ) h--; }
```

Commentary: The behavior of the above implementation may not match the behavior of the C++ stack library. To ensure a segmentation fault in top(), when the stack is empty one may use the following code. `if( empty() ) return *(int*)0; else return S[t];`

स्टैक के लिए कोड(भर जाने पर विस्तार) (Implementing stack (expanding when full))

```
private:
  void expand() {
    int new_size = N*2; // हमने इस वृद्धि को लैब में देखा था!! (We observed the growth in our lab!!)
    int* tmp = (int*) malloc( sizeof(int)*new_size );//नया ऐरे (New array)
    for( unsigned i =0; i < N; i++ ) {// पुराने ऐरे से कॉपी करें (copy from the old array)
      tmp[i] = S[i];
    }
    free(S);        // पुरानी मेमोरी छोड़ें (Release old memory)
    S = tmp;        // लोकल फ़ील्ड्स को अपडेट करें (Update local fields)
    N = new_size; //
  }
};
```

Exercise 3.3
क्या हम प्रोसेसर्स की DMA (Direct Memory Access) सुविधा का उपयोग करके कॉपी कर सकते हैं?
(Can we utilize the DMA (Direct Memory Access) feature of processors to perform the copy?)

कॉम्प्लेक्सिटी (Efficiency)

सभी ऑपरेशन $O(1)$ में किए जाते हैं अगर स्टैक में कोई विस्तार नहीं होता है।
(All operations are performed in $O(1)$ if there is no expansion to the stack capacity.)

विस्तार की कीमत क्या है? (What is the cost of expansion?)

Topic 3.3

क्यों एक्सपोनेंशियल विस्तार रणनीति? (Why exponential growth strategy?)

विस्तार रणनीति (Growth strategy)

हम विस्तार के लिए दो संभावित विकल्पों पर विचार करें। (Let us consider two possible choices for growth.)

- कांस्टेंट विस्तार (Constant growth): `new_size = N + c` (किसी कांस्टेंट c के लिए (for some fixed constant c))
- एक्सपोनेंशियल विस्तार (Exponential growth): `new_size = 2*N`

ऊपर के दोनों में से कौन सा बेहतर है? (Which of the above two is better?)

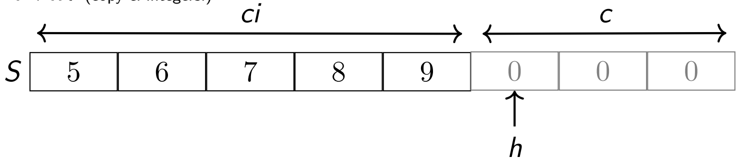
कांस्टेंट विस्तार का विश्लेषण (Analysis of constant growth)

हम मान लेते हैं कि शुरुआत में $N = 0$ है और n एक के बाद एक pushes हैं।
(Let us suppose initially $N = 0$ and there are n consecutive pushes.)

हर cth push के बाद, एक विस्तार होगा। (After every cth push, there will be an expansion operation.)

इसलिए, $(ci + 1)$th पुश पर विस्तार (Therefore, the expansion operation at $(ci + 1)$th push will)

- $c(i + 1)$ आकार की मेमोरी को आवंटित करेगा और (allocate memory of size $c(i + 1)$ and)
- ci इंटीजर्स कॉपी करेगा। (copy ci integers.)



ith विस्तार की कीमत (Cost of ith expansion): $c(2i + 1)$.

Commentary: We are assuming that allocating memory of size k costs k time, which may be more efficient in practice. Bulk memory copy can also be sped up by vector instructions.

कांस्टेंट विस्तार का विश्लेषण(2) (Analysis of constant growth(2))

n pushes के लिए, n/c विस्तार होंगे। (For n pushes, there will be n/c expansions.)

सभी विस्तारों की कुल लागत: (The total cost of expansions:)

$$c(1 + 3 + ... + (2\frac{n}{c} + 1)) = c(n/c)^2 \in O(n^2)$$

नॉनलीनियर लागत! (Nonlinear cost!)

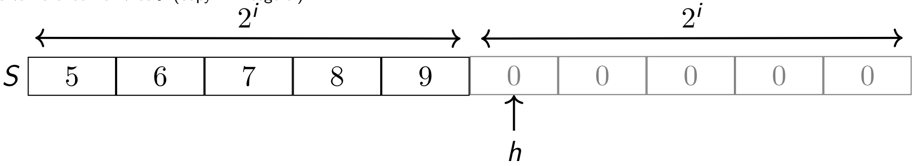
एक्सपोनेंशियल विस्तार का विश्लेषण (Analysis of exponential growth)

हम मान लेते हैं कि शुरुआत में $N = 1$ है और $n = 2^r$ एक के बाद एक pushes हैं।
(Let us suppose initially $N = 1$ and there are $n = 2^r$ consecutive pushes.)

विस्तार केवल $(2^i + 1)$वें पुश पर होंगे, जहां $i \in [0, r-1]$ है।
(The expansion operations will only occur at the $(2^i + 1)$th push, where $i \in [0, r-1]$.)

$2^i + 1$वें पुश पर विस्तार (The expansion operation at $2^i + 1$th push will)

- 2^{i+1} आकार की मेमोरी आवंटित करेगा और (allocate memory of size 2^{i+1} and)
- 2^i इंटीजर्स कॉपी करेगा। (copy 2^i integers.)



विस्तार की लागत: (Cost of the expansion:) $3 * 2^i$.

एक्सपोनेंशियल विस्तार का विश्लेषण(2) (Analysis of exponential growth(2))

2^r pushes के लिए, अंतिम विस्तार $2^{r-1} + 1$ पर होगा। (For 2^r pushes, the last expansion would be at $2^{r-1} + 1$.)

सभी विस्तारों की कुल लागत: (The total cost of expansions:)

$$3(2^0 + ... + 2^{r-1}) = 3 * (2^r - 1) = 3 * (n - 1)$$

लीनियर लागत! push की औसत लागत $O(1)$ बनी रहती है। (Linear cost! The average cost of push remains $O(1)$.)

Exercise 3.4

डबल क्यों? तिगुना क्यों नहीं? 1.5 गुना क्यों नहीं? विस्तार के मान को चुनने के लिए किसी और बारे में भी सोचना पड़ेगा?
(Why double? Why not triple? Why not 1.5 times? Is there a trade-off?)

Commentary: The policy is not the same in all the libraries for vectors. What is the policy in the following implementations of STL?
GCC STL: (libstdc++) `https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_vector.h#L2187`
Clang STL: (libc++) `https://github.com/llvm/llvm-project/blob/main/libcxx/include/__vector/vector.h#L909`
MASC STL: `https://github.com/microsoft/STL/blob/main/stl/inc/vector#L2006`

Topic 3.4

स्टैक के प्रयोग (Applications of stack)

स्टैक हर जगह हैं (Stacks are everywhere)

स्टैक एक मूलभूत डेटा स्ट्रक्चर है। (Stack is a foundational data structure.)

ये बहुत सारी अल्गोरिथ्म्स में इस्तेमाल होता है। (It shows up in a vast range of algorithms.)

उदाहरण: परेंथेसिस मिलाना (Example: matching parentheses)

समस्या: (Problem:)
एक टेक्स्ट दिया गया है, जांचें कि क्या उस-में मिलान वाले परेंथेसिस हैं।
(Given an input text, check if it has matching parentheses.)

उदाहरण: (Examples:)

- "{a[sic]tik}" ✓
- "{a[sic}tik}" ✗

```
bool parenMatch(string text ) {
  std::stack<char> s;
  for(char c : text ) {
    if( c == '{' or c == '[' ) s.push(c);
    if( c == '}' or c == ']' ) {
      if( s.empty()       ) return false;
      if( c-s.top() != 2 ) return false;
      s.pop();
    }
  }
  if( s.empty() ) return true;
  return false;
}
```

Topic 3.5

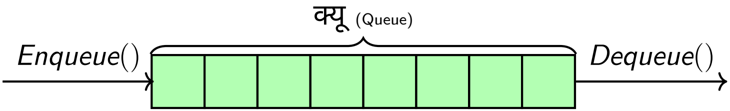
क्यू (Queue)

क्यू (Queue)

Definition 3.2

क्यू एक कंटेनर है जहां एलिमेंट्स को पहले-आने-पहले-निकाले (FIFO) क्रम के अनुसार जोड़ा और हटाया जाता है।
(Queue is a container where elements are added and deleted according to the first-in-first-out (FIFO) order.)

- जोड़ना enqueue कहलाता है। (Addition is called enqueue)
- हटाना dequeue कहलाता है। (Deleting is called dequeue)



Example 3.2

- हवाई अड्डे में प्रवेश (Entry into an airport)
- इमारत में लिफ्ट को बुलाना (थोड़ा अलग है; यह एक प्राथमिकता कतार है)
(Calling a lift in a building (slightly different; it is a priority queue))

क्यू का इंटरफेस (Interface of queue)

Reference: https://en.cppreference.com/w/cpp/container/queue

क्यू इंटरफ़ेस में चार मुख्य फंक्शन हैं। (Queue supports four main interface methods)

- `queue<T> q` : नयी क्यू q को आवंटित करता है। (allocates a new queue q.)
- `q.enqueue(e)` : एलिमेंट e को क्यू के अंत में जोड़ता है। (Adds the given element e to the end of the queue.) (push)
- `q.dequeue()` : क्यू से पहले एलिमेंट को हटाता है। (Removes the first element from the queue.) (pop)
- `q.front()` : पहले एलिमेंट को पढ़ें। (access the first element .)

कुछ सहायक फ़ंक्शन (Some support functions)

- `q.empty()` : जाँच करता है कि क्या क्यू खाली है। (checks whether the queue is empty.)
- `q.size()` : एलिमेंट्स की संख्या वापस करता है। (returns the number of elements.)

Commentary: All literature uses the terms enqueue and dequeue, but unfortunately, the C++ library uses push for enqueue and uses pop for dequeue. Other languages, such as Java, use the terms enqueue and dequeue.

क्यू के axioms* (Axioms of queue*)

1. `queue<T> q; Assert(q.empty() == true);`

2. `q.enqueue(e); Assert(q.empty() == false);`

3. ```
   Assume(q.empty() == true);
   q.enqueue(e); Assert(q.front() == e);
   ```

4. ```
   Assume(q.empty() == false && old_q == q);
   q.enqueue(e); Assert(old_q.front() == q.front());
   ```

5. ```
   Assume(q.empty() == true && old_q == q);
   q.enqueue(e); q.dequeue(); Assert(old_q == q);
   ```

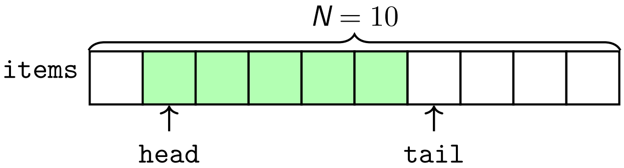
6. ```
   Assume(q.empty() == false && q == q1);
   q.enqueue(e);q.dequeue(); q1.dequeue();q1.enqueue(e); Assert(q == q1);
   ```

Topic 3.6

एरे-आधारित क्यू (Array implementation of queue)

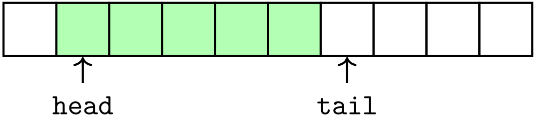
ऐरे-आधारित क्यू (Array-based implementation)

- क्यू को एक वृत्ताकार ढंग से एक ऐरे `items` में संग्रहीत किया जाता है। (Queue is stored in an array `items` in a circular fashion)
- तीन इन्टिजर्स क्यू की स्थिति को दर्ज करते हैं (Three integers record the state of the queue)
  1. `N` का संकेत करता है कि कतार की उपलब्ध क्षमता (N-1) है (`N` indicates the available capacity (N-1) of the queue)
  2. `head` क्यू के सामने की स्थिति को दर्शाता है। (`head` indicates the position of the front of the queue)
  3. `tail` क्यू के पिछले हिस्से के एक के बाद की स्थिति को दर्शाता है। (`tail` indicates position one after the rear of the queue)

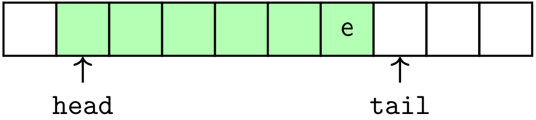


एरे पर Enqueue ऑपरेशन (Enqueue operation on the array)

मान लें की क्यू निम्नलिखित स्तिथि मैं है। (Consider the following state of the queue.)

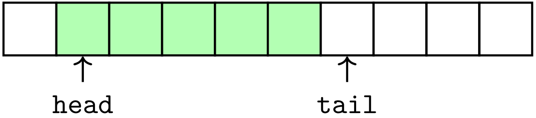


enqueue(e) ऑपरेशन के बाद: (After the enqueue(e) operation:)

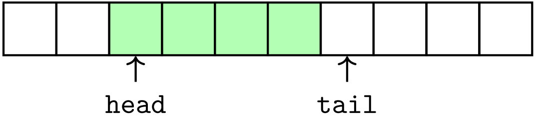


एरे पर Dequeue ऑपरेशन (Dequeue operation on the array)

मान लें की क्यू निम्नलिखित स्तिथि मैं है। (Consider the following state of the queue.)



`dequeue()` ऑपरेशन के बाद: (After the `dequeue()` operation:)

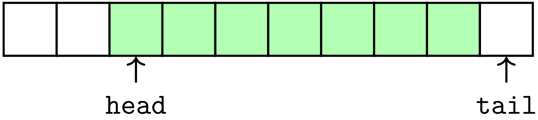


Exercise 3.5

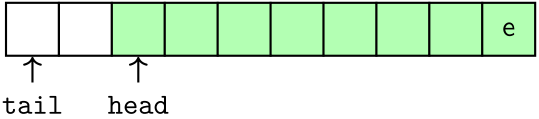
1. `front()` कहाँ से पढ़ेगा? (Where will `front()` read from?)
2. क्यू का आकार क्या है? (What is the size of the queue?)

अधिकांश ऐरे का उपयोग करने के लिए घूम कर वापस आएं (Wrap around to utilize most of the array)

मान लें की क्यू निम्नलिखित स्तिथि में है। (Consider the following state of the queue.)

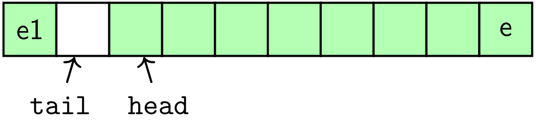


`enqueue(e)` ऑपरेशन के बाद, हम `tail` को 0 में स्थानांतरित करते हैं। (After the `enqueue(e)` operation, we move the `tail` to 0.)



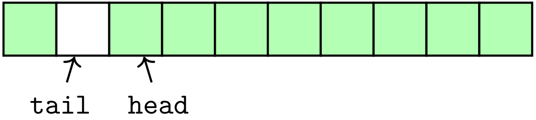
यह हमें ऐरे को बार-बार उपयोग करने देता है। (Wrap-around allows us to use the array repeatedly.)

एक और `enqueue(e1)` ऑपरेशन के बाद: (After another `enqueue(e1)` operation:)

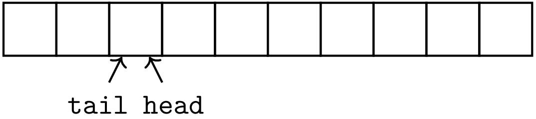


पूर्ण और खाली क्यू (Full and empty queue)

पूर्ण क्यू: (Full queue:)



खाली क्यू: (Empty queue:)



Exercise 3.6

क्या हम सभी N जगहों का उपयोग कर सकते हैं? (Can we use all N cells for storing elements?)

ऐरे कोड (Array implementation)

कोड सी++ में लिखा नहीं गया है; हम धीरे-धीरे स्लाइड्स पर अव्यवस्था से बचने के लिए सुडो कोड की ओर बढ़ेंगे।
(The code is not written in C++; We will slowly move towards pseudo code to avoid clutter on slides.)

```
int head = 0, tail=0, N = INITIAL_CAPCITY;

Object items[N];                    // प्रारंभिक आकार (Some initial size)

bool empty()   { return (head == tail);  }

int size()     { return (N+tail-head)%N; }

Object front() { return items[head];     }
```

ऐरे कोड(2) (Array implementation(2))

```
void dequeue() {
  if( empty() ) throw Empty;                   //क्यू खाली है (Queue is empty)
  free(items[head]);items[head] = NULL; //मेमोरी हटाएं (Clear memory)
  head = (head+1)%N;       //Remove an element
}

void enqueue( Object x ) {
  if ( size() == N-1 ) expand();//क्यू पूर्ण होने पर विस्तार करें (expand when queue is full)
  items[tail] = x;
  tail = (tail+1)%N; //एलिमेंट डालें (insert element)
}
```

Exercise 3.7

हमारे स्टैक में, हमने pop में फ्री को कॉल नहीं किया, लेकिन हमने dequeue में फ्री को कॉल किया। क्यों?

(In our stack implementation, we did not invoke free in pop, but we invoked free in dequeue. Why?)

Commentary: In the stack implementation, we were only handling a stack of int. Here, we are handling a queue of **arbitrary objects**. If the object has a dynamic size, then it cannot live on the queue itself. We will store a reference to the object on the queue and allocate the object somewhere else. Therefore, we must free the objects on dequeue (the above syntax of free is not in C++; It will only work when items[head] is a reference). However, there is no need to free an int because it has a fixed size (32 bits $\leq$ size of pointers=64bits) and it was stored on the array of the stack itself.

Topic 3.7

लिंक्ड लिस्ट के माध्यम से क्यू (Queue via linked lists)

क्या हम विस्तार से बच सकते हैं? (Can we avoid expansion?)

जब भी एरे पूर्ण होते हैं, उन्हें विस्तार की आवश्यकता होती है। (Arrays need expansion whenever they are full.)

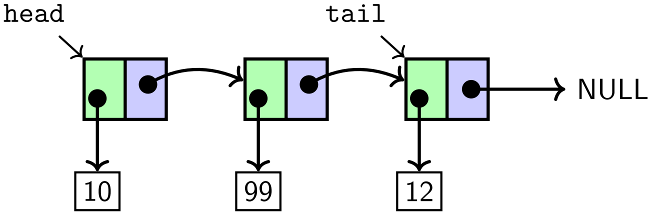
क्या हम कॉपी करने की आवश्यकता के बिना विस्तार कर सकते हैं? (Can we expand without the need for copying?)

हम इसे प्राप्त करने के लिए लिंक्ड लिस्ट का उपयोग कर सकते हैं। (We may use linked lists to achieve this.)

लिंक्ड लिस्ट्स (Linked lists)

Definition 3.3

एक लिंक्ड लिस्ट में नोड्स (मेमोरी के छोटे टुकड़े) होते हैं जिनमें दो फील्ड होते हैं: data और next पॉइंटर। नोड्स next पॉइंटर के माध्यम से एक श्रृंखला बनाते हैं। data पॉइंटर्स उन ऑब्जेक्ट्स की ओर इशारा करते हैं जो लिंक्ड लिस्ट पर होते हैं।

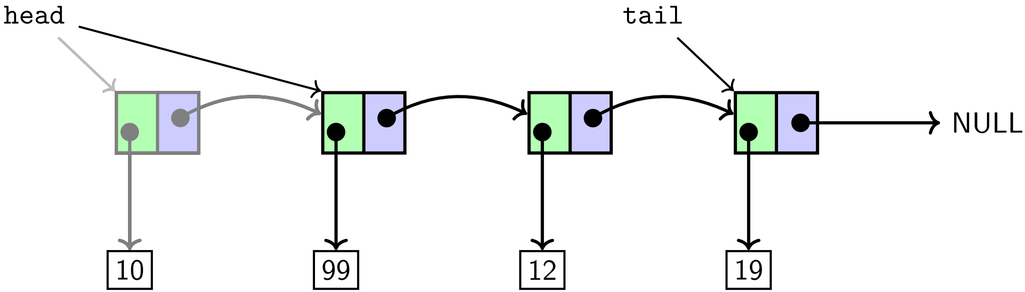


Exercise 3.8

a. यदि हम एक क्यू के लिए एक लिंक्ड लिस्ट का उपयोग करते हैं, तो क्यू का सामने कौन सा नोड होना चाहिए?
(If we use a linked list for implementing a queue, which side should be the front of the queue?)

b. एक ऐरे और लिंक्ड लिस्ट के बीच मूलभूत अंतर क्या है? (What is the fundamental difference between an array and a linked list?)

लिंक्ड लिस्ट में Dequeue (Dequeue in linked lists)

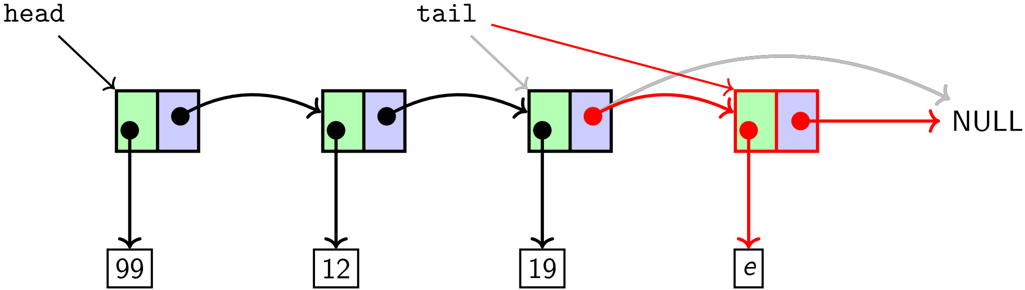


Exercise 3.9

10 को रखने वाले ऑब्जेक्ट के साथ क्या होता है? (What happens to the object containing 10?)

Commentary: Answer: There are several choices. The object is deallocated, the reference to the object is returned to the caller, or the copy of the object is returned to the caller. Different implementations may do it differently. This behavior is not part of the specification of the queue.

लिंक्ड लिस्ट में Enqueue(e) (Enqueue(e) in linked lists)



Exercise 3.10

a. कौन बेहतर है: एरे या लिंक्ड लिस्ट? (Which one is better: array or linked list?)

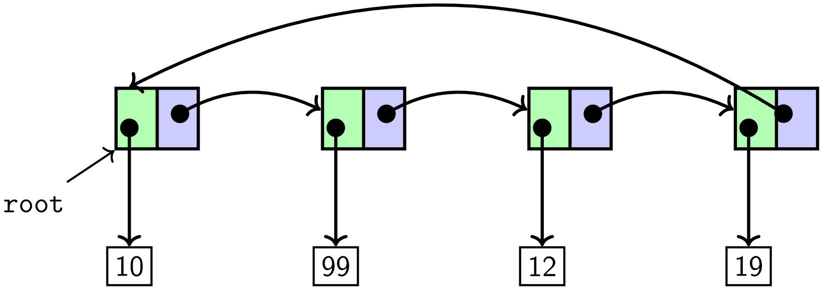
b. क्या हमें टेल पॉइंटर की आवश्यकता है? (Do we need the tail pointer?)

Topic 3.8

सर्कुलर लिंक्ड लिस्ट (Circular linked list)

सर्कुलर लिंक्ड लिस्ट (Circular linked list)

Definition 3.4
एक सर्कुलर लिंक्ड लिस्ट में, नोड्स `next` पॉइंटर के माध्यम से एक सर्कुलर चेन बनाते हैं।
(In a circular linked list, the nodes form a circular chain via the `next` pointer.)



एक `root` पॉइंटर सर्कुलर लिस्ट के किसी नोड की ओर इशारा करता है। (A `root` pointer points at some node of the circular list.)

सर्कुलर लिंक्ड लिस्ट के माध्यम से क्यू (Queue via circular linked lists)

हम क्यू के लिए सर्कुलर लिंक्ड लिस्ट का उपयोग कर सकते हैं। (We may use the circular linked list for queue.)

क्यू के लिए, एक अकेला `root` पॉइंटर `head` और `tail` का काम कर सकता है।
(For queue, a single `root` pointer can do the job of the `head` and `tail`.)

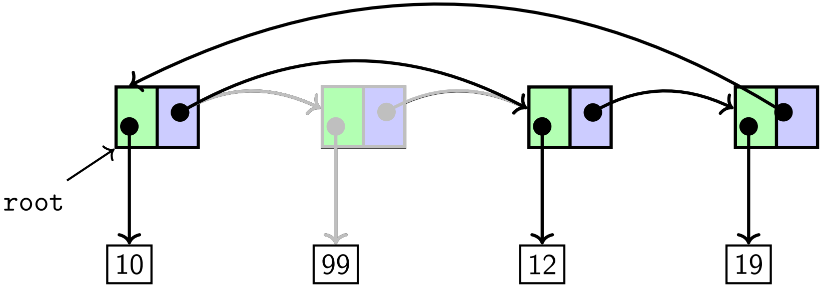
- `tail = root`
- `head = root->next`

Exercise 3.11

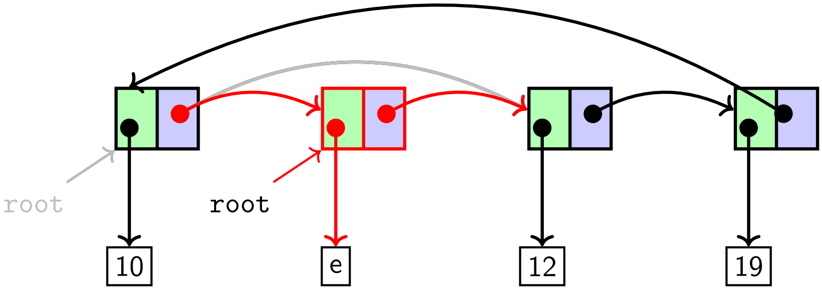
`front()` द्वारा कौन सा एलिमेंट वापस किया जाना चाहिए? (Which element should be returned by `front()`?)

Commentary: Please note that we cannot implement queue if we try using `root` as `head` and `root->next` as `tail`.

सर्कुलर लिंक्ड लिस्ट में dequeue (Dequeue in circular linked lists)



सर्कुलर लिंक्ड लिस्ट में enqueue(e) (enqueue(e) in circular linked lists)



Exercise 3.12
सर्कुलर लिंक्ड लिस्ट का उपयोग करके क्यू का सुडो कोड दें।
(Give pseudo code of the implementation of queue using a circular linked list.(Midsem 2023))

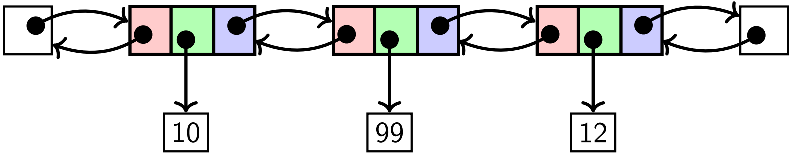
Topic 3.9

दोहरी लिंक्ड लिस्ट और Deque (Doubly linked list and Deque)

दोहरी लिंक्ड लिस्ट (Doubly linked lists)

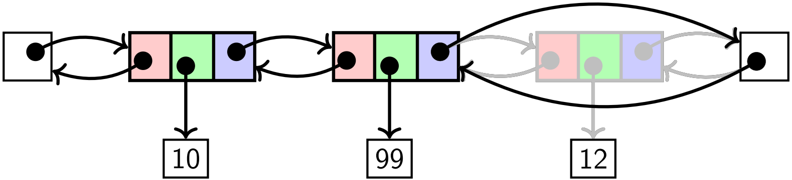
Definition 3.5

एक दोहरी लिंक्ड लिस्ट के नोड्स में तीन फ़ील्ड्स होते हैं: prev, data, और next पॉइंटर। नोड्स prev और next पॉइंटर्स के माध्यम से एक द्विदिशीय श्रृंखला बनाते हैं। data पॉइंटर्स उन ऑब्जेक्ट्स की ओर इशारा करते हैं जो लिंक्ड लिस्ट पर होते हैं।



दोनों सिरों पर, दो सेंटिनेल नोड्स किसी भी डेटा को नहीं रखते हैं और ये सूची के समापन के संकेत हैं। (At both ends, two dummy or sentinel nodes do not store any data and are marker of termination of the list.)

एक दोहरी लिंक्ड लिस्ट में एक नोड को हटाना (Deleting a node in a doubly linked list)



Exercise 3.13

एक नोड को डालने के लिए कदम दें। (Give steps to insert a node.)

Deque (Double-ended queue)

Definition 3.6

Deque एक कंटेनर है जहां एलिमेंट्स को अंतिम-अंदर-पहले-बाहर (LIFO) और पहले-अंदर-पहले-बाहर (FIFO) क्रम के अनुसार जोड़ा और हटाया जाता है। (Deque is a container where elements are added and deleted according to both last-in-first-out (LIFO) and first-in-first-out (FIFO) order.)

Deque का इंटरफेस (Interface of Deque)

Reference: `https://en.cppreference.com/w/cpp/container/deque`

Deque इंटरफ़ेस में छः प्रमुख फ़ंक्शन होते हैं। (Deque supports six main interface methods)

- `deque<T> q` : एक नया Deque q को आवंटित करता है। (allocates a new deque q.)
- `q.push_back(e)` : पीछे एलिमेंट e को जोड़ता है। (Adds element e to the back.)
- `q.push_front(e)` : सामने एलिमेंट e को जोड़ता है। (Adds element e to the front.)
- `q.pop_front()` : पहला एलिमेंट हटाता है। (Removes the first element.)
- `q.pop_back()` : अंतिम एलिमेंट को हटाता है। (Removes the last element.)
- `q.front()` : पहले एलिमेंट को पढता है। (accesses the first element.)
- `q.back()` : अंतिम एलिमेंट को पढता है। (accesses the last element.)

अन्य सहायक फ़ंक्शन क्यू के समान होते हैं। (Other support functions are similar to queue.)

हम Deque को दोहरी लिंक्ड लिस्ट का उपयोग करके लागू कर सकते हैं।
(We can implement the Deque data structure using the doubly linked list.)

deque के माध्यम से स्टैक और क्यू (Stack and queue via Deque)

हम दोनों स्टैक और क्यू को `Deque` के इंटरफेस का उपयोग करके लागू कर सकते हैं।
(We can implement both the stack and the queue using the interface of the deque.)

Exercise 3.14

- `Deque` के कौन से फंक्शन्स स्टैक के लिए इस्तेमाल कर सकते हैं? (Which functions of deque implement a stack?)
- `Deque` के कौन से फ़ंक्शन क्यू के लिए इस्तेमाल कर सकते हैं? (Which functions of deque implement a queue?)

सभी परिवर्तन करने वाले फंक्शन्स $O(1)$ हैं। (All modification operations are implemented in $O(1)$.)

Exercise 3.15

क्या हम `size` को $O(1)$ में एक दोहरी लिंक्ड सूची में कर सकते हैं? (Can we implement `size` in $O(1)$ in a doubly linked list?)

Topic 3.10

Tutorial problems

Exercise: Use of stack

Exercise 3.16

The span of a stock's price on the *ith* day is the maximum number of consecutive days (up to the *ith* day) the price of the stock has been less than or equal to its price on day *i*.

Example: for the price sequence $2\ 4\ 6\ 3\ 5\ 7$ of a stack, the span of prices is $1\ 2\ 3\ 1\ 2\ 6$.

Give a linear-time algorithm that computes s_i for a given price series.

Exercise: flipping Dosa

Exercise 3.17

There is a stack of dosas on a tava, of distinct radii. We want to serve the dosas of increasing radius. Only two operations are allowed: (i) serve the top dosa, (ii) insert a spatula (flat spoon) in the middle, say after the first k, hold up this partial stack, flip it upside-down, and put it back. Design a data structure to represent the tava, input a given tava, and produce an output in sorted order. What is the time complexity of your algorithm?
This is also related to the train-shunting problem.

Exercise: exponential growth

Exercise 3.18

a. Analyze the performance of exponential growth if the growth factor is three instead of two. Does it give us better or worse performance than the doubling policy?

b. Can we do a similar analysis for growth factor 1.5?

Exercise: reversing a linked list (Quiz 2024)

Exercise 3.19

Give an algorithm to reverse a linked list. You must use only three extra pointers.

Exercise: middle element

Exercise 3.20

Give an algorithm to find the middle element of a singly linked list.

Exercise: stack and queue (Endsem 2023)

Exercise 3.21

Given two stacks $S1$ and $S2$ (working in the LIFO method) as black boxes, with the regular methods: "Push", "Pop", and "isEmpty", you need to implement a Queue (specifically : Enqueue and Dequeue working in the FIFO method). Assume there are n Enqueue/ Dequeue operations on your queue. The time complexity of a single method, Enqueue or Dequeue, may be linear in n, however the total time complexity of the n operations should also be $\Theta(n)$.

Topic 3.11

Problems

True or False

Exercise 3.22

Mark the following statements as True/False and also justify.

1. The element at the top of the stack has been in the stack for the longest time.
2. A stack does not allow random access to its elements.
3. O(1) is the worst-case time complexity of dequeue in a queue containing m elements implemented using an array.
4. The pop operation on a stack is always $O(1)$.
5. A stack can be implemented using a doubly-linked list as well as a singly-linked list
6. We can traverse a singly linked list backward, starting from the last node to the first node.
7. In C++, `std::queue<T>` has method `push_back`.
8. In C++ `std::vector<T>`, the buffer size is doubled whenever there is a call to `push_back` after the vector is full.

Problem: messy queue

Exercise 3.23

The mess table queue problem: There is a common mess for k hostels. Each hostel has some N_1,...,N_k students. These students line up to pick up their trays in the common mess. However, the queue is implemented as follows: If a student sees a person from his/her hostel, she/he joins the queue behind this person. This is the ”enqueue” operation. The ”dequeue” operation is, as usual, at the front. Think about how you would implement such a queue. What would be the time complexity of enqueue and dequeue? Do you think the average waiting time in this queue would be higher or lower than that of a normal queue? Would there be any difference in any statistic? If so, what?

Merge sorted linked lists (Quiz 2023)

Exercise 3.24

Write a time and space-efficient algorithm to merge k sorted-linked lists in sorted order, each containing the same no of elements?

Exercise: axioms of queue**

Exercise 3.25

Using axioms of queue, show that the assert in the following does not fail.

```
queue<int> q,q1;
q.enqueue(2);
q.enqueue(0);
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(3);

q1.enqueue(7);
q1.enqueue(3);
Assert(q == q1);
```

Topic 3.12

Extra slides: Discussion

Point of discussion: vector instructions for fast expansion*

Exercise 3.26

Save the following code in `copy.cpp` and compile with the following options.

```
void copy(int* dst, int* src) {
  for( unsigned i =0; i < 32; i++ ) {
    dst[i] = src[i];
  }
}
```

- Compile with: `g++ copy.cpp -O3 -S`
- Compile with: `g++ copy.cpp -O3 -mavx512f -S` (Enables vector instructions!)

Read the assembly generated in the `copy.s` file. Observe the difference.

Point of discussion: store values or references in the collection*

A collection like a queue has two choices: either store values or references to the values.

Operations on the collection need to behave differently. For example,

Operation	Storing values	Storing references
`enqueue(Object e)`	store a copy of the e	store a reference to e
`front()`	return a copy	return a reference
`dequeue()`	delete the object	remove the reference

Exercise 3.27
What are the advantages or disadvantages of the methods of storing collections?

Point of discussion: Where are the libraries in my computer?

Exercise 3.28

How does a C++ compiler find the libraries? Where are they stored in a computer?

End of Lecture 3