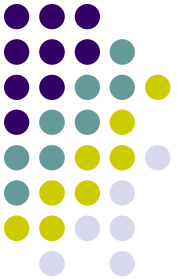
NIT



Структуре података (и алгоритми)

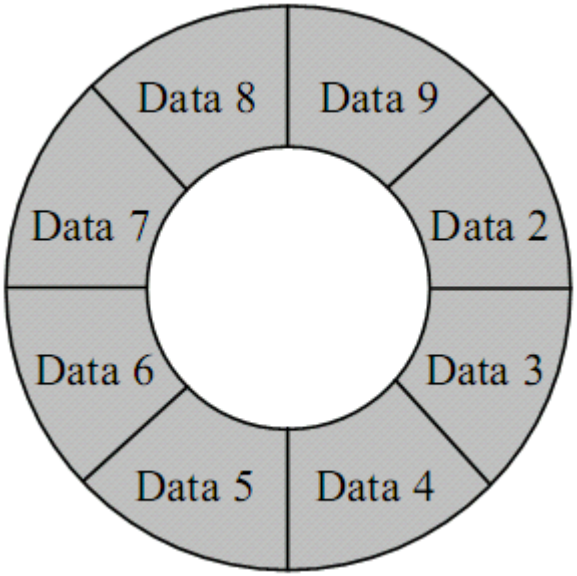
NIT

Бафер

- **Бафер** је структура података чија је улога да ублажи, или чак елиминише, потребу за синхронизацијом између различитих активности. Може се посматрати као прихватна меморија у коју неке активности уписују резултате (произвођач) док неке друге активности те податке читају (потрошач). Дакле, бафер је посредник између произвођача и потрошача. Назив се често употребљава и за било какво парче меморије (јер у суштини увек постоји неки „произвођач” и неки „потрошач”).
- Сваки бафер садржи одређени број поља за смештање информација. Број тих поља је величина бафера.
- Поља могу бити било шта.

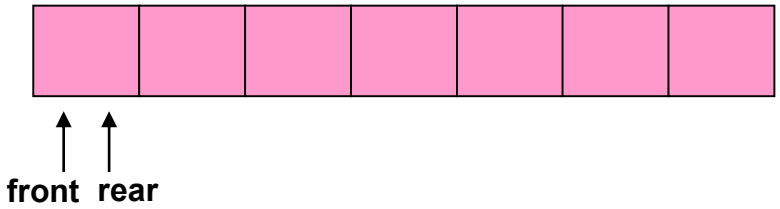
1. Кружни бафер

- **Кружни бафер (енгл. circular/ring buffer)** је бафер који је организован тако да су, сликовито гледано, његова поља поређана у круг.
- Обично се реализује како FIFO структура и тада поседује два показивача која одређују где се подаци могу уписивати (реп) и одакле се могу читати (глава).
- Предност овако организованог бафера је у томе што нема померања елемената, већ само мењања по једног показивача, дакле постављање и вађење података су јефтине радње.
- Може се десити да произвођач препише податке које потрошач још није прочитао.
  - Ако нам је динамика система позната, онда може бити довољно само направити бафер довољне величине.
  - Ако нам динамика система није позната, или је променљива, онда се морају користити механизми синхронизације. Ипак, тада је синхронизација лабавија него без бафера.

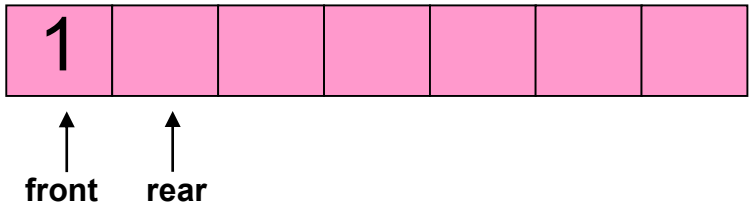


Пример рада са кружним бафером

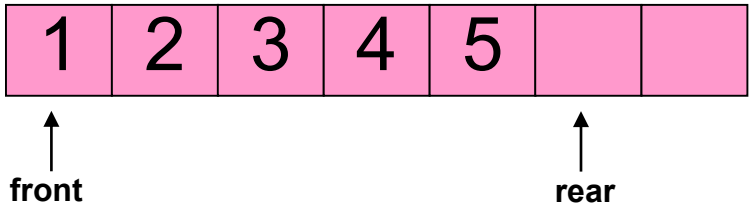
1. **Бафер је празан**



2. **Убаци 1**

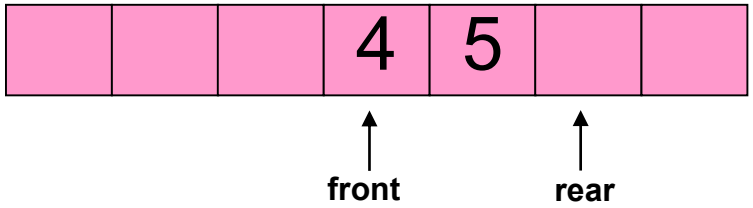


3. **Убаци 2, 3, 4 и 5**

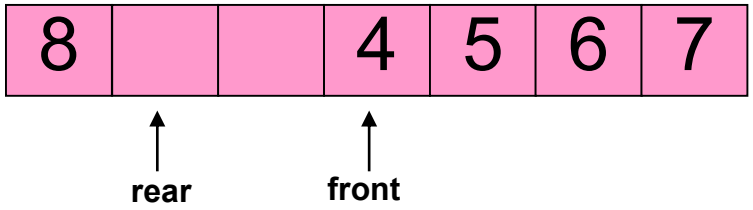


Пример рада са кружним бафером

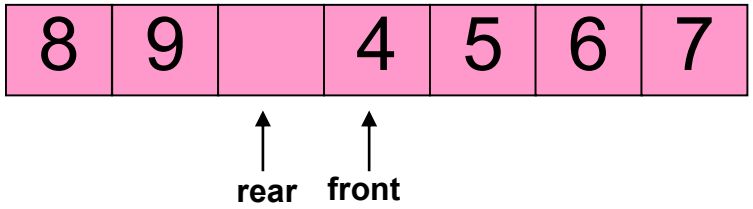
4. Вади 1, 2 и 3



5. Убаци 6, 7 и 8

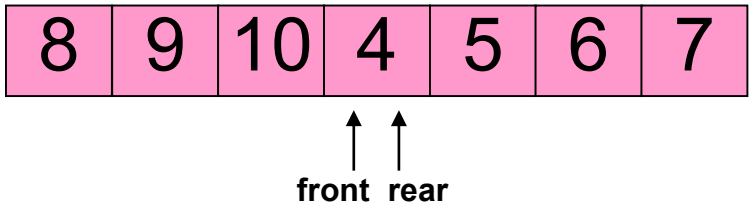


6. Убаци 9 => Уколико нам је потребан информација о попуњености бафера онда овај случај може представљати стање скроз попуњеног бафера

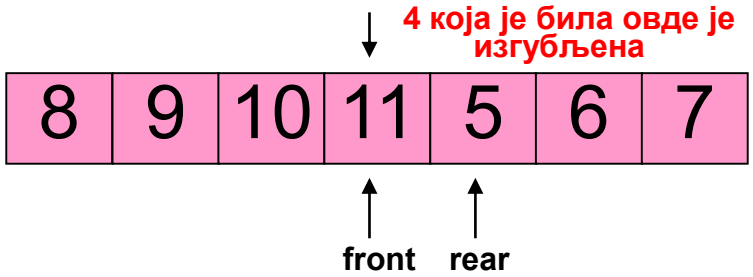


Пример рада са кружним бафером

7. Убаци 10 => Да ли је бафер скроз попуњен или скроз празан? Ако уведемо трећу променљиву која ће о томе водити рачуна можемо тек сад прогластити бафер скроз пуним



8. Убаци 11 => Прекорачење.



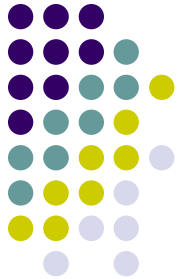
Три варијанте кружног бафера

- Ако нам није потребна информација о попуњености
  Само сам бафер и два показивача.
- Ако нам јесте потребна информација о попуњености
  - Без коришћења помоћне променљиве

    И даље само бафер и два показивача, али једно поље у баферу мора увек бити празно. Једноставна имплементација и минимум трошка приликом уписа и читања.
  - Са коришћењем помоћне променљиве

    Мора се увести додатна променљива. Предност је поптуна искоришћеност бафера, али је мана већи трошак уписа и читања, јер се мора освежавати и та додатна променљива

NIT



2. Дупли бафер

- Дупли бафер је организација бафера која је погодна за ситуације када потрошач чита из једног блока док произвођач пише у други блок података
- Врло је сродан кружном баферу. У принципу представља кружни бафер величине 2, где је једно поље бафера у ствари низ неких мањих поља (па се и за тај низ мањих поља исто примењује назив „бафер” - отуд назив „дупли бафер”)

NIT

Употреба бафера

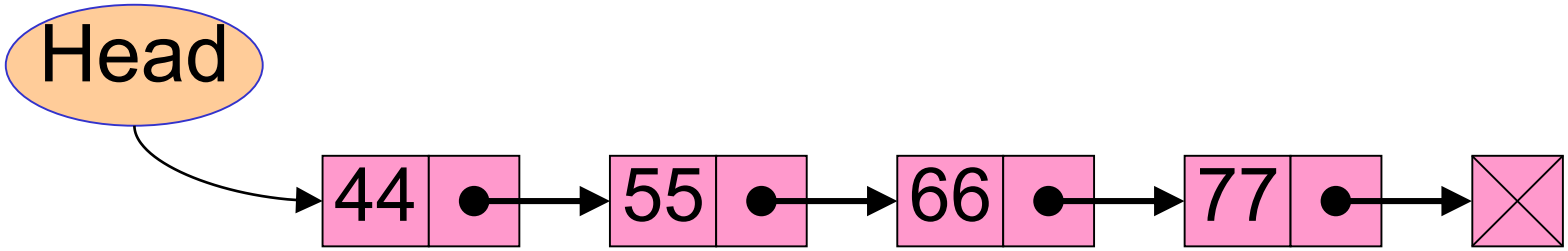
- Бафери, посебно кружни бафери и њихова специјална варијанта дупли бафери, користе се врло често у системској програмској подршци, али и у програмској подршци за наменске системе.
- Посредством кружних бафера комуницирају језгра на вишејезгарским процесорима, процеси са било којом периферијом, и слично.
- Заправо, често најважнији, а некада и једини, елемент реалног времена у наменским системима директно су везани за коришћење бафера.

Рецимо, крајњи рок за обављање задатка формирања наредне слике је тренутак када видео спрега почне да чита из тог поља у дуплом баферу. Ако задатак закасни, видео спрега ће прочитати претходне податке и изгледаће као да слика стоји.

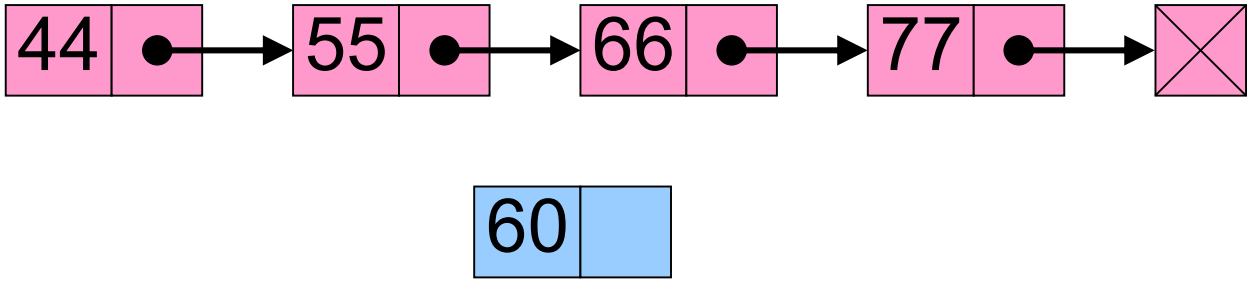
NIT

3. Повезана листа

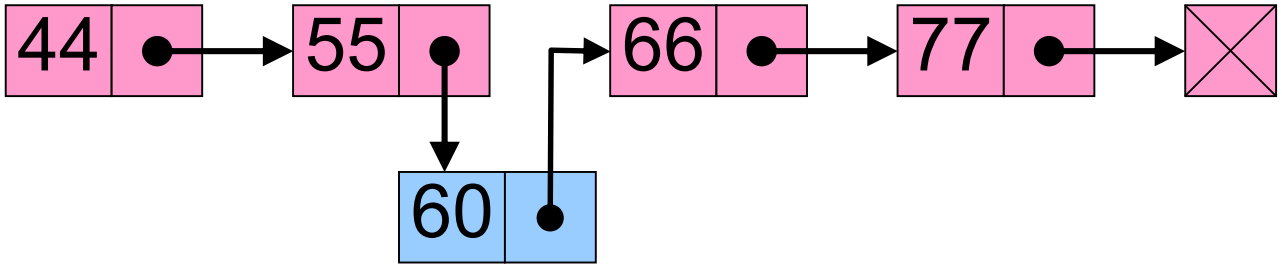
- Повезана листа је структура података код које:
  - Суседни елементи су повезани показивачима
  - Последњи елемент показује на NULL
  - Може расти и смањивати се динамички на релативно јефтин начин
  - Ограничена је само величином слободне меморије
  - Не троши много више меморије него што је потребно за саме податке



Убацивање елемента

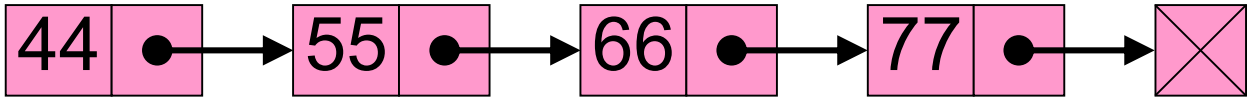


Node to insert

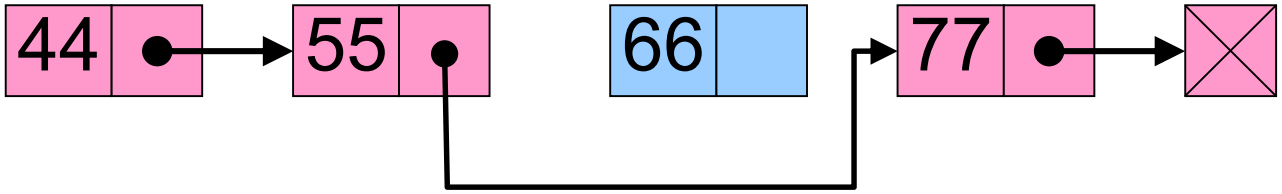


Node to insert

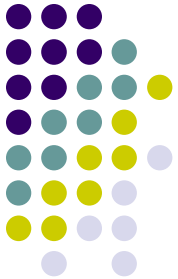
Брисање елемента



Node to delete



NIT

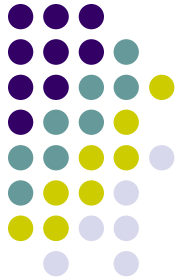


Укратко

- Убацивање елемента:
  - Направи се (заузме се меморија, динамички) нови елемент и попуни одговарајућим подацима
  - Показивач новог елемента се поставља да показује на следећи елемент у листи
  - Показивач елемента који претходи новом елементе поставља се да показује на њега
- Брисање елемента:
  - Показивач елемента претходника се поставља да показује на следбеника елемента који се брише
  - Меморија заузета за обрисани елемент се ослобађа

NIT

Низ наспрам повезане листе

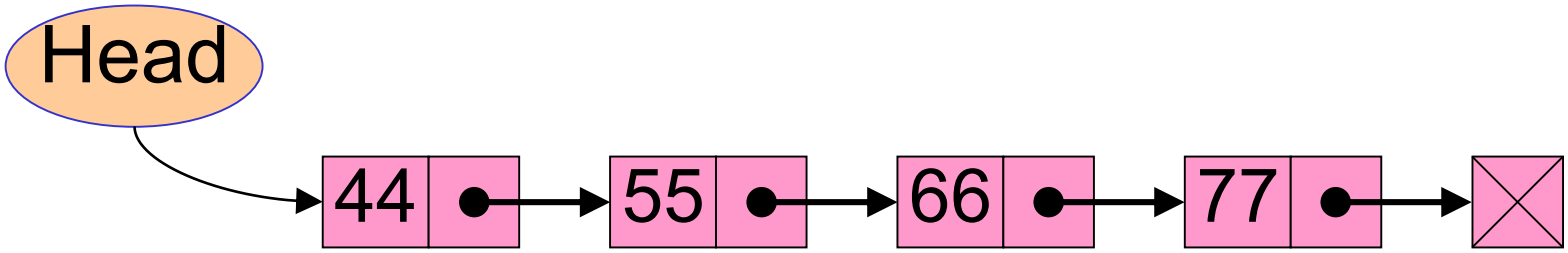


- Низови су погодни за:
  - Произвољан приступ елементима (путем индекса)
  - Додавање и брисање елемената на крају
- Повезане листе су погодне за:
  - Додавање и брисање елемента било где у листи
  - Примене где је секвенцијалан приступ довољан
  - Када број елемената није могуће предвидети у напред, или када је потребно радити са изузетно много елеманата (Питање: чиме је ограничена величина низа?)

Врсте повезаних листа

Једноструко повезане листе

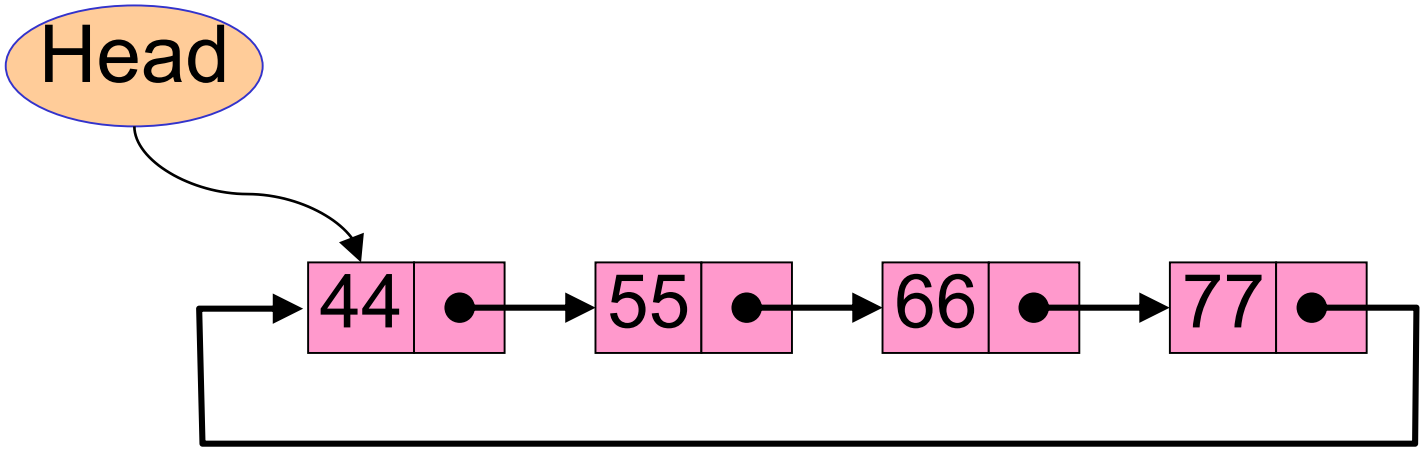
Само један показивач по елементу. Показује на наредни елемент. Последњи показује на NULL.



Врсте повезаних листа

Кружне једноструко повезане листе

Показивач последњег елемента показује поново на први

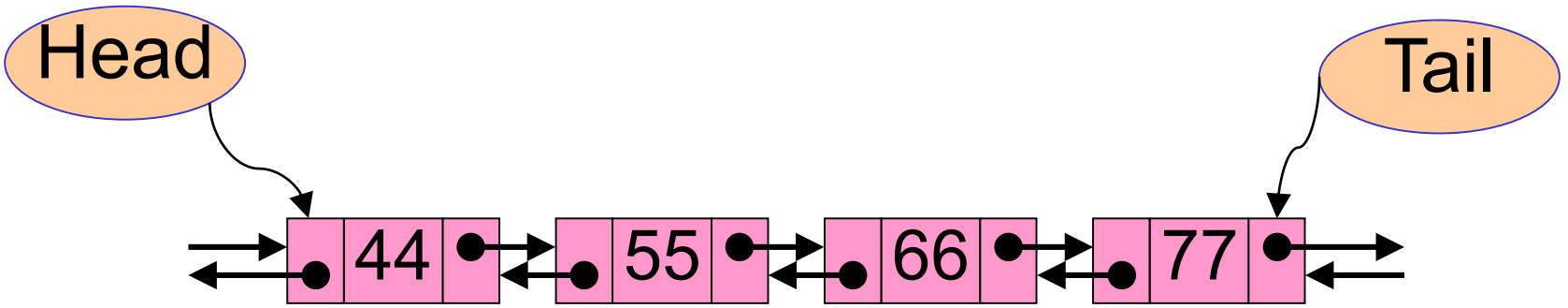


NIT

Врсте повезаних листа

Двоструко повезане листе

- Два показивача по елементу - један показује на наредни, један на претходни елемент.
- Лако кретање у оба смера
- Сада осим показивача на главу постоји и показивач на реп.



Имплементација листе

- За листу је довољан само показивач који показује на први елемент. Ако он показује на NULL, листа је празна
- Прављење елемента

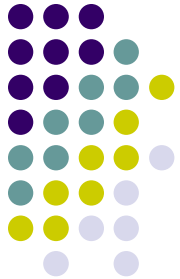
```
struct node
{
   int   id;
   char  name[25];
   int   age;
   struct node* next;
};
```

```
struct person {
  int id;
  char name[25];
  int age;
};

struct node {
  struct person data;
  struct node* next;
};
```

```
head = (struct node*)malloc(sizeof(struct node));
```

- За сваки елемент листе следеће ствари се морају урадити:
  - **Заузимање меморије**
  - **Постављање поља елемента**
  - **Измена показивача елемената на одговоарајућим местима**

NIT



Пролазак кроз листу

- Једноставна операција која се своди на следеће:
  - Крени од главе (или репа, у случају двоструко повезане листе)
  - Прати показиваче на наредни елемент (или претходни, ако код двоструко повезане листе пролазимо у назад)
  - Заустави се када дођеш до NULL показивача

    (Када се зауставити у случају кружне листе?)

NIT

Додавање новог елемента

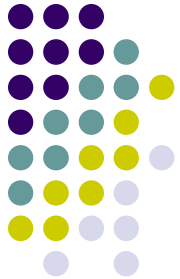
- **Додавање на почетак**
  - Показивач новог елемента се поставља да показује на први елемент листе (оно на шта показује глава).
  - Глава се мења да показује на нови елемент.
- **Додавање негде у средину**
  - Нови да показује на наредни
  - Претходни елемент да показује на нови
- **Додавање на крај**
  - Показивач новог елемента да буде NULL
  - Показивач последњег елемента поставља се да показује на нови елемент

За додавање елемента у једноструко спрегнуту листу потребно је добавити елемент (или главу) иза којег желимо нови елемент да додамо.

Брисање елемента аналогно додавању.

NIT

Додавање новог елемента



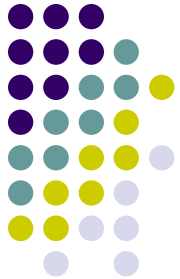
```
struct node {
  struct person data;
  struct node* next;
};
struct node* List;

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node* place, struct node* element) {
  element->next = place->next;
  place->next = element;
}
void addEleAtEnd(struct node* last, struct node* element) {
  element->next = NULL;
  last->next = element;
}
```

Додавање новог елемента

```
struct node {
  struct person data;
  struct node* next;
};
struct node* List;

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node* place, struct node* element) {
  element->next = place->next;
  place->next = element;
}
void addEleAtEnd(struct node* last, struct node* element) {
  element->next = NULL; // last->next == NULL
  last->next = element;
}
```

NIT

Додавање новог елемента



```
struct node {
  struct person data;
  struct node* next;
};
struct node* List;

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node* place, struct node* element) {
  element->next = place->next;
  place->next = element;
}

addEleInFront(&List, ele);
addEle(pos, ele);
```

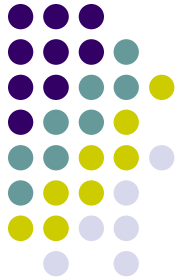
NIT

Додавање новог елемента

```
struct node {
  struct person data;
  struct node* next;
};
struct node* List;

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node** nextAddr, struct node* element) {
  element->next = *nextAddr;
  *nextAddr = element;
}

addEle(&List, ele);
addEle(&(pos->next), ele);
```

NIT

Додавање новог елемента

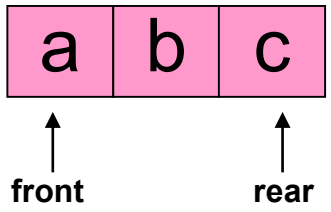


```
struct node {
  struct person data;
  struct node* next;
};
struct node List; // <-

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node* place, struct node* element) {
  element->next = place->next;
  place->next = element;
}

addEle(&List, ele);
addEle(pos, ele);
```

NIT

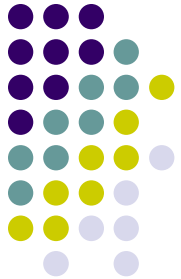
4. Ред

- Ред је у суштини структура података код које податак који се први упише бива први и прочитан (FIFO).
- Ред може бити имплементиран на било који начин.
- Да би нешто било ред довољно је да може садржати више података истог типа и да им се приступа у FIFO маниру.
- Теоријски гледано ред је неограничене величине. Пошто у пракси то није могуће ово се своди на чињеницу да у дефиницији реда није специфицирано шта се дешава ако се максимална величина реда прекорачи.
- Са друге стране ред би требао да има информацију томе да ли је празан, да би се спречило преузимање елемената из празног реда.



NIT

Ред - имплементација коришћењем низа



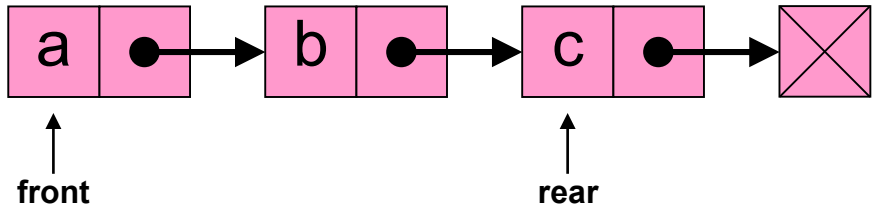
- То је у ствари кружни бафер!

  Једино пошто је неопходна информација о томе да ли је ред празан не може се користити изведба без информације о попуњености (које изведбе се могу користити?)

  И нема више шта даље.

Ред - имплементација коришћењем повезане листе

- Једноструко повезана листа је довољна за имплементацију реда.
- Елементи се додају на крај листе, а скидају са почетка.
- Једини проблем је чињеница да за додавање на крај морамо добавити последњи елемент листе.
- Зато је згодно увести додатни показивач који ће показивати на последњи елемент у листи (да га не добављамо стално)

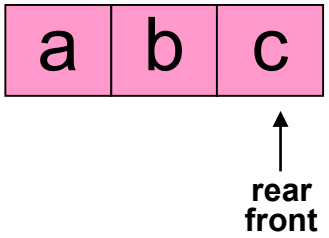


- Због тога поступак за додавање елемента на крај листе постаје за трунку сложенији

NIT

5. Стек

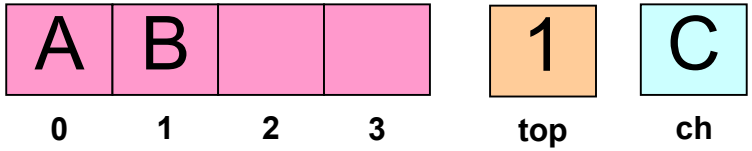
- Стек је обрнуто од реда. Податак који се последњи упише бива први прочитан (LIFO).
- И стек може бити имплементиран на било који начин.
- И стек у теорији има бесконачну величину, али у пракси нема, па и овде имамо недефинисано понашање у случају прекорачења максималне величине стека.
- И стек мора детектовати када је празан, да би се спречило читање.



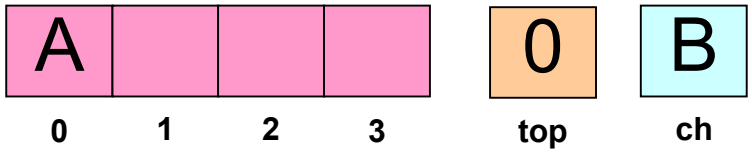
Стек – имплементација коришћењем низа

Уз низ нам је потребна променљива која ће одређивати где је тренутно врх стека:

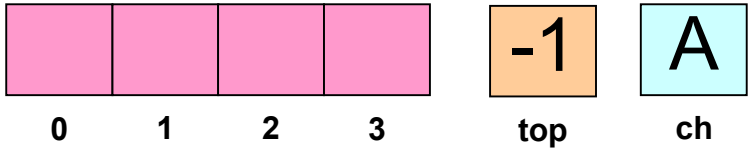
```
ch = Pop(stack)
```




```
ch = Pop(stack)
```




```
ch = Pop(stack)
```

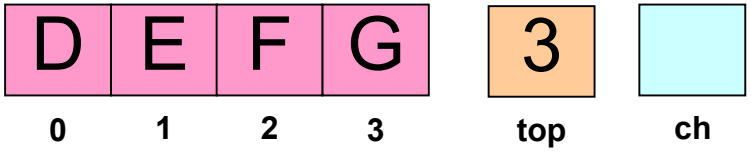


Стек је празан када је индекс **-1**.
Које акције треба обавити при стављану елемента на стек, а које при скидању?

Стек – имплементација коришћењем низа

Шта се дешава у случају прекорачења максималне величине стека?

```
Push(stack, 'D')
Push(stack, 'E')
Push(stack, 'F')
Push(stack, 'G')
```



И затим покушамо ово: **`Push(stack, 'H')`**

Стек - имплементација коришћењем повезане листе

Једноструко повезана листа је погодна и за имплементацију стека

Да ли је боље стављати и скидати елементе са почетка или са краја?