ASD—5
LISTY

1 Podstawowe operacje na listach

Lista jest to skończony ciąg elementów, $L = [x_1, x_2, \ldots, x_n]$. Skrajne elementy listy x_1 i x_n nazywają się końcami listy (odpowiednio, lewym i prawym), $|L| = n$ nazywa się długością (lub rozmiarem) listy. Szczególnym przypadkiem listy jest lista pusta $L = []$. Do podstawowych operacji na listach należą:

- `Locate(x,L)`:
  zwraca wskaźnik do elementu x na liście L. Jeśli x nie ma na liście, to zwracana jest specjalna wartość NULL.

- `Retrieve(p,L)`:
  zwraca element z pozycji p. Jeśli takiej pozycji nie ma na liście ($p > |L|$), to zwracana jest specjalna wartość NULL. Szczególnymi przypadkami operacji `Retrieve` są `Front(L)` i `Rear(L)`, które zwracają, odpowiednio, pierwszy i ostatni element z listy.

- `Insert(x,p,L)`:
  wstaw x na pozycję p na liście L. Jeśli $p > |L|+1$, to wynik operacji jest nieokreślony. Szczególnymi przypadkami operacji `Insert` są `Push(x,L)` i `Inject(x,L)`, które polegają na wstawieniu x, odpowiednio, na początek i koniec listy L.

- `Delete(p,L)`:
  usuń p-ty element z listy L. Jeśli $p > |L|$, to wynik operacji jest nieokreślony. Szczególnymi przypadkami operacji `Delete` są `Pop(L)` i `Eject(x,L)`, które polegają na usunięciu, odpowiednio, pierwszego i ostatniego elementu z listy L.
- `MakeNull(L)`:
  utwórz pustą listę L.

Listę, na której wykonywane są operacje Inject, Front i Pop nazywamy kolejką, natomiast listę z operacjami Front, Push i Pop nazywamy stosem.

2 Dowiązaniowa implementacja listy

2.1 Listy jednokierunkowy

Oto przykład definicji struktury węzła listy

```
struct node {
  long value; /* dane przechowywane w wezle */
  struct node *next;
};
```

W przypadku listy jednokierunkowej dostęp do listy uzyskujemy przez wskaźnik do jej pierwszego węzła.

```
struct node *first = NULL;
```

`first = NULL` oznacza, że lista jest pusta. Oto implementacja poszczególnych operacji na liście jednokierunkowej.

```
struct node{
  long value;
  struct node *next;
};

void MakeNull(struct node **L)
{
  *L = NULL;
}

struct node *Locate(long x, struct node *L)
{
  while (L != NULL && L->value != x)
    L = L->next;
  return L;
}

struct node *Retrieve(int p, struct node *L)
/* p >= 1 */
{
  while (L != NULL && p > 1){
    L = L->next;
    p--;
  }
  if (p == 1)
    return L;
  else
    return NULL;
}
```

```
void Insert(long x, int p, struct node **L)
/* p >= 1 */
{
  struct node *q, *s, *t;

  s = NULL;
  t = *L;
  while (t != NULL && p > 1)
  {
     s = t;
     t = t->next;
     p--;
   }

   if (p == 1)
   {
      q = malloc(sizeof(struct node));
      q->value = x;
      q->next = t;
      if (s == NULL)
        *L = q;
      else
        s->next = q;
    }
}

void Delete(int p, struct node **L)
{
```

```
  struct node *s, *t;

  s = NULL;
  t = *L;
  while (t != NULL && p > 1)
  {
     s = t;
     t = t->next;
     p--;
   }
   if (p == 1 && t != NULL)
   {
     if (s == NULL)
       *L = t->next;
     else
       s->next = t->next;
     free(t);
    }
}
```

Oto przykład listowej implementacji sita Eratostenesa.

```
#include <stdio.h>

struct node{
  long value;
  struct node *next;
};

void BuildList(struct node **, int);
void Sieve(struct node *);
```

```
void PrimeNumbers(struct node *);

main()
{
  int n;
  struct node *Lista;

  printf("Podaj n > 1: ");
  scanf("%d", &n);

  BuildList(&Lista, n);
  Sieve(Lista);
  PrimeNumbers(Lista);

  return 0;
}

void BuildList(struct node **L, int n)
{
  struct node *p, *q;

  q = NULL;
  while (n >= 2)
  {
    p = malloc(sizeof(struct node));
    p->value = n;
    p->next = q;
    q = p;
    n--;
```

```
  }
  *L = q;
}
void Sieve(struct node *L)
{
  struct node *p, *q, *r;
  long i;

  for (p = L; p != NULL; p = p->next){
    i = p->value;
    r = p;
    q = p->next;
    while (q != NULL)
      if (q->value % i == 0){
        q = q->next;
        free(r->next);
        r->next = q;
      }
      else{
        r = q;
        q = q->next;
      }
  }
}


void PrimeNumbers(struct node *L)
{
  for (; L != NULL; L = L->next)
```

```
    printf("%d\n",L->value);
}
```

Oto jeszcze jeden przykład użycia list: SORTOWANIE NA LISTACH

```
#include <stdio.h>

#define Bool int

struct node{
  long value;
  struct node *next;
};

struct queue_item{
  struct node *lista;
  struct queue_item *qnext;
};

struct queue{
  struct queue_item *first, *last;
};

struct node *NewList(long);
struct node *MergeLists(struct node *, struct node *);
void BuildEmptyQueue(struct queue *);
void Inject(struct node *, struct queue *);
struct node *Pop(struct queue *);
Bool OneItem(struct queue);
void PrintList(struct node *);
```

```
main()
{
  int n;
  long x;
  struct queue Q;

  BuildEmptyQueue(&Q);
  printf("Podaj n >= 1: ");
  scanf("%d", &n);

  for (;n > 0; n--){
    printf("Kolejny element ciag: ");
    scanf("%d", &x);
    Inject(NewList(x), &Q);
  }

  while (! OneItem(Q))
    Inject(MergeLists(Pop(&Q),Pop(&Q)),&Q);

  PrintList(Q.first->qnext->lista);

  return 0;
}

struct node *NewList(long x)
{
  struct node *p;
```

```
  p = malloc(sizeof(struct node));
  p->value = x;
  p->next = NULL;

  return p;
}

struct node *MergeLists(struct node *L1,
                        struct node *L2)
{
  if (L1 == NULL)
    return L2;
  else if (L2 == NULL)
    return L1;
  else if (L1->value < L2->value){
    L1->next = MergeLists(L1->next,L2);
    return L1;
  }
  else{
    L2->next = MergeLists(L1,L2->next);
    return L2;
  }
}

void BuildEmptyQueue(struct queue *Q)
{
  Q->first = malloc(sizeof(struct queue_item));
  Q->last = Q->first;
```

```
}

void Inject(struct node *L, struct queue *Q)
{
  Q->last->qnext = malloc(sizeof(struct queue_item));
  Q->last = Q->last->qnext;
  Q->last->lista = L;
}

struct node *Pop(struct queue *Q)
{
  struct queue_item *p;
  struct node *v;

  p = Q->first;
  Q->first = Q->first->qnext;
  v = Q->first->lista;
  free(p);
  return v;
}

Bool OneItem(struct queue Q)
{
   return (Q.first != Q.last)
           &&
           (Q.first->qnext == Q.last);
}

void PrintList(struct node *L)
```

```
{
  while (L != NULL){
    printf("%d\n",L->value);
    L = L->next;
  }
}
```