Kurs C

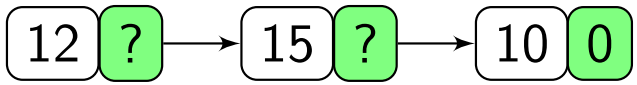
Radosław Łukasik

Wykład dodatkowy: listy i drzewa

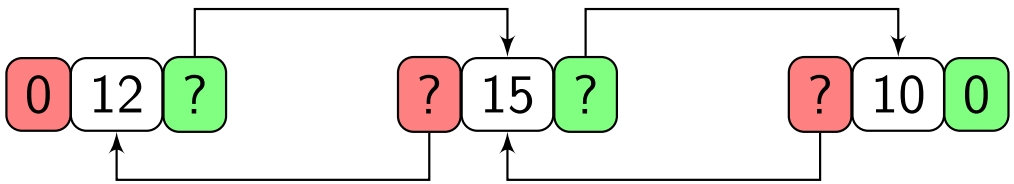
Listy

Lista jest ciągiem elementów (węzłów) zawierającym pewne dane oraz wskaźnik (lub wskaźniki) na element następny (oraz poprzedni).
Rozróżniamy trzy rodzaje list:

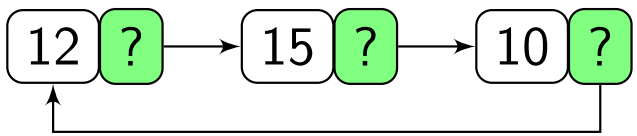
- jednokierunkowa - występuje tylko wskaźnik na element następny, ostatni element zawiera wskaźnik zerowy;



- dwukierunkowa - występują oba wskaźniki (na następny i poprzedni), przy czym wskaźnik na poprzedni pierwszego elementu i wskaźnik na następny ostatniego elementu są zerowe;



- cykliczna - tak jak jednokierunkowa, przy czym ostatni element zawiera wskaźnik na pierwszy element listy.



Aby operować na listach potrzebujemy zawsze wskaźnika na pierwszy element listy.

Listy można zastosować do utworzenia innych struktur danych jak np. stos czy kolejka.
Stos - liniowa struktura danych, na której elementy przetwarzane są w kolejności od tego, który pojawił się najpóźniej do tego, który był wpisany na samym początku. Poszczególne elementy można przeglądać, ale aby pobrać element znajdujący się poniżej, trzeba pobrać ze stosu wszystkie elementy znajdujące się nad nim. W algorytmach stos reprezentowany jest przez strukturę LIFO (Last In First Out).
Kolejka jest strukturą działającą przeciwnie do stosu. Dane przetwarzane są w kolejności ich pojawienia się, tzn. począwszy od tego, który dopisany był na samym początku, do tego który znalazł się w kolejce na samym końcu. W algorytmach kolejka reprezentowana jest przez strukturę FIFO (First In First Out).

Przykład listy dwukierunkowej

```
typedef struct student{
    char imie[32];
    char nazwisko[32];
    unsigned ocena;
    unsigned indeks;
    struct student* prev;
    struct student* next;
} student;
student * pierwszy=0;// wskaźnik na początek listy
```

Dodawanie nowego elementu do listy

Jeżeli chcemy dodać do listy nowy element, to oprócz dodania właściwej zawartości elementu należy ustawić odpowiednio wskaźniki na następny i poprzedni element w przypadku listy dwukierunkowej. Wstawiając nowy element pomiędzy dwa istniejące musimy m.in. w elemencie poprzednim zmienić wskaźnik na następny. Stąd w przypadku list jednokierunkowej i cyklicznej najłatwiej wstawić nowy element jako drugi lub na początek (tylko w przypadku listy jednokierunkowej). W przypadku listy dwukierunkowej wstawianie w dowolnym miejscu jest jednakowo proste. Aby uprościć wstawianie w listach jednokierunkowych lub cyklicznych można przechowywać wskaźnik na aktualną pozycję w liście by wstawiać za nią kolejny element. Jeśli chcielibyśmy dodawać nowy element przed danym elementem, to musielibyśmy najpierw znaleźć jego poprzednik, co w przypadku list jednokierunkowych i cyklicznych ma średnią złożoność liniowo zależną od długości listy.

Przykład

```
student *dodajza(student *aktualny){
    student *nowy=(student) malloc(sizeof(student));
    if(nowy==0){
        printf("Nie mozna dodac studenta. Za malo pamieci.\n");
        nowy=aktualny;
    }else{
        printf("Podaj imie: ");
        scanf(" %s",nowy->imie);
        printf("Podaj nazwisko: ");
        scanf(" %s",nowy->nazwisko);
        printf("Podaj ocene: ");
        scanf(" %u",&(nowy->ocena));
        printf("Podaj indeks: ");
        scanf(" %u",&(nowy->indeks));
/*-->*/ nowy->prev=aktualny;// bez tego w listach jednokierunkowych
        if(aktualny==0){// pusta lista?
            nowy->next=0;
        }else{
            nowy->next=aktualny->next;
            aktualny->next=nowy;
/*-->*/       if(nowy->next !=0) (nowy->next)->prev=nowy;
        }// if powyżej tylko w listach dwukierunkowych
    }
    return nowy;
}
```

Przykład

```
void *dodajnapoczatek(student **poczatek){
    student *nowy=(student*) malloc(sizeof(student));
    if(nowy==0){
        printf("Nie mozna dodac studenta. Za malo pamieci.\n");
    }else{
        printf("Podaj imie: ");
        scanf(" %s",nowy->imie);
        printf("Podaj nazwisko: ");
        scanf(" %s",nowy->nazwisko);
        printf("Podaj ocene: ");
        scanf(" %u",&(nowy->ocena));
        printf("Podaj indeks: ");
        scanf(" %u",&(nowy->indeks));
/*-->*/ nowy->prev=0;// bez tego w listach jednokierunkowych
        if(*poczatek==0){// pusta lista?
            nowy->next=0;
        }else{
            nowy->next=*poczatek;
/*-->*/       (*poczatek)->prev=nowy;// bez tego w listach jednokier.
        }
        *poczatek=nowy;// ustawiamy nowy początek
    }
}
```

Przykład

```
enum POZYCJA{PRZED =1, PO,  POCZ, KON, SORT};
student *dodaj(student **poczatek,student *aktualny,enum POZYCJA pos){
    student *nowy=(student*) malloc(sizeof(student));
    int temp;
    if(nowy==0){
        printf("Nie mozna dodac studenta. Za malo pamieci.\n");
        return *poczatek;
    }else{
        printf("Podaj imie: ");
        scanf(" %s",nowy->imie);
        printf("Podaj nazwisko: ");
        scanf(" %s",nowy->nazwisko);
        printf("Podaj ocene: ");
        scanf(" %u",&(nowy->ocena));
        printf("Podaj indeks: ");
        scanf(" %u",&(nowy->indeks));
        nowy->prev=0;
        nowy->next=0;
        if(*poczatek==0){// pusta lista?
            *poczatek=nowy;
        }else{// dla niepustej listy
            if(aktualny==0 || (pos!=PRZED && pos!=PO))
                aktualny=*poczatek;
            switch(pos){// wybieramy miejsce dodania
```

Przykład c.d.

```
case SORT:// lista jest posortowana alfabetycznie
    while(1){
        temp=strcmp(aktualny->nazwisko,nowy->nazwisko);
        if(temp>0) break;
        if(temp==0 &&strcmp(aktualny->imie,nowy->imie)>=0)
            break;
        if(aktualny->next==0) break;
        aktualny=aktualny->next;
    }
    if(temp<0)// wstawiamy po lub przed
        goto KONIEC;
case POCZ:
case PRZED:
    nowy->next=aktualny;
    nowy->prev=aktualny->prev;
    if(aktualny->prev==0)
        *poczatek=nowy;
    else
        (aktualny->prev)->next=nowy;
    aktualny->prev=nowy;
    break;
```

Przykład c.d.

```
            case KON:
                while(aktualny->next !=0)// szukamy ostatniego
                    aktualny=aktualny->next;
            case PO:
              KONIEC:
                nowy->next=aktualny->next;
                nowy->prev=aktualny;
                if(aktualny->next !=0)
                    aktualny->next->prev=nowy;
                aktualny->next=nowy;
                break;
            }
        }
        return nowy;
    }
}
```

Przykład wcześniejszy ale dla list jednokierunkowych

```
typedef struct studjk{
    char imie[32];
    char nazwisko[32];
    unsigned ocena;
    struct studjk* next;
} studjk;

enum POZYCJA{PRZED =1, PO,  POCZ, KON, SORT};
student *dodaj(studjk **poczatek,studjk *aktualny,enum POZYCJA pos){
    studjk *stemp=0,*nowy=(studjk*) malloc(sizeof(studjk));
    int temp;
    if(nowy==0){
        printf("Nie mozna dodac studenta. Za malo pamieci.\n");
        return *poczatek;
    }else{
        printf("Podaj imie: ");
        scanf(" %s",nowy->imie);
        printf("Podaj nazwisko: ");
        scanf(" %s",nowy->nazwisko);
        printf("Podaj ocene: ");
        scanf(" %u",&(nowy->ocena));
        printf("Podaj indeks: ");
        scanf(" %u",&(nowy->indeks));
        nowy->next=0;
```

Przykład c.d.

```
        if(*poczatek==0){// pusta lista?
            *poczatek=nowy;
        }else{// dla niepustej listy
            if(aktualny==0 || (pos!=PRZED && pos!=PO))
                aktualny=*poczatek;
            switch(pos){// wybieramy miejsce dodania
            case SORT:// lista jest posortowana alfabetycznie
                while(1){
                    temp=strcmp(aktualny->nazwisko,nowy->nazwisko);
                    if(temp>0) break;
                    if(temp==0 &&strcmp(aktualny->imie,nowy->imie)>=0)
                        break;
                    if(aktualny->next==0) break;
                    stemp=aktualny;
                    aktualny=aktualny->next;
                }
                if(temp<0)// wstawiamy po
                    goto KONIEC;
/*-->*/           else{// lub przed
/*-->*/               if(stemp) stemp->next =nowy;
/*-->*/               nowy->next =aktualny;
/*-->*/           }
/*-->*/           break;
```

Przykład c.d.

```
            case POCZ:
            case PRZED:
                nowy->next=aktualny;
                if(aktualny==*poczatek)
                    *poczatek=nowy;
                else{
                    temp=*poczatek;// szukamy poprzednika
/*-->*/             while(stemp->next != aktualny)
/*-->*/                 stemp=stemp->next;
/*-->*/             stemp->next=nowy;
                }
                break;
            case KON:
                while(aktualny->next !=0)// szukamy ostatniego
                    aktualny=aktualny->next;
            case PO:
              KONIEC:
                nowy->next=aktualny->next;
                aktualny->next=nowy;
                break;
            }
        }
        return nowy;
    }
}
```

Usuwanie elementu z listy

Jeżeli chcemy usunąć dany element listy, to musimy również zmienić w poprzednim elemencie wskaźnik na następny, a w kolejnym elemencie wskaźnik na poprzedni (dla listy dwukierunkowej). W listach dwukierunkowych jest to proste do zrobienia. W listach jednokierunkowych łatwo się usuwa pierwszy element listy. W pozostałych przypadkach musimy zawsze wyszukać element poprzedzający usuwany i zmienić odpowiedni wskaźnik.
Jeżeli chcemy usunąć wszystkie elementy, to zaczynamy od pierwszego zapisujemy wskaźnik na kolejny element i usuwamy dany. Czynność tą powtarzamy aż do końca listy lub w w przypadku listy cyklicznej do otrzymania wskaźnika początkowego.

Przykład

```
void usun(student *aktualny,student **poczatek){
    if(aktualny!=0){
        if(aktualny->prev !=0)
            (aktualny->prev)->next=aktualny->next;
        else// usuwamy pierwszy element listy
            *poczatek=aktualny->next;// zmieniamy wskaźnik początek
        if(aktualny->next !=0)
            (aktualny->next)->prev=aktualny->prev;
        free(aktualny);
        printf("Usunieto\n");
    }else
        printf("Lista pusta\n");
}
```

```
void usunliste(student **poczatek){
    while(*poczatek!=0){
        student *temp=(*poczatek)->next;
        free(*poczatek);
        *poczatek=temp;
    }
    printf("Lista usunieta\n");
}// kod dobry dla list jedno i dwukierunkowych
```

Przykład - usuwanie jednego elementu w listach jednokierunkowych

```
typedef struct listajedn{
    char imie[32];
    char nazwisko[32];
    unsigned ocena;
    struct listajedn* next;
} listajedn;

void usun(listajedn *aktualny,listajedn **poczatek){
    if(aktualny!=0){
        if(aktualny == *poczatek)//usuwanie początku listy
            *poczatek=aktualny->next;
        else{
            listajedn *poprzedni=*poczatek;// szukamy poprzednika
            while(poprzedni->next != aktualny)
                poprzedni=poprzedni->next;
            poprzedni->next=aktualny->next;
        }
        free(aktualny);
        printf("Usunieto\n");
    }else
        printf(")Lista pusta\n");
}// średnia złożoność O(n)
```

Wyszukiwanie elementu o podanych kryteriach

Gdy chcemy wyszukać element z listy, który spełnia pewne nasze kryteria, to możemy te kryteria oznaczyć jako bity pewnej wartości (flagi), którą będziemy przekazywali do funkcji wyszukującej. Można tutaj użyć typu wyliczeniowego **enum** do utworzenia wszystkich możliwych rozłącznych parametrów wyszukiwania. W ten sposób suma odpowiednich flag mówiłaby nam jakie parametry uwzględniamy w wyszukiwaniu. W samym wyszukiwaniu, czy też przed jego wywołaniem utworzylibyśmy dodatkowy element tego samego typu jak na liście (ale lepiej na stosie, wtedy nie musimy martwić się o jego usunięcie) i wypełnilibyśmy go tymi danymi, które są oznaczone w naszych flagach.

Przykład

```
enum fstudent{imie =1, nazwisko =2, ocena =4, indeks=8};//2^i
student *szukaj(student *pocz,student *dane,unsigned flagi){
    if(dane!=0){
        while(pocz!=0){
            if(flagi & imie)
                if(strcmp(dane->imie,pocz->imie)!=0){
                    pocz=pocz->next; continue;
                }
            if(flagi & nazwisko)
                if(strcmp(dane->nazwisko,pocz->nazwisko)!=0){
                    pocz=pocz->next; continue;
                }
            if(flagi & ocena)
                if(dane->ocena!=pocz->ocena){
                    pocz=pocz->next; continue;
                }
            if(flagi & indeks)
                if(dane->indeks!=pocz->indeks){
                    pocz=pocz->next; continue;
                }
            break;
        }
    }
    return pocz;
}
```

Operacje na wszystkich elementach listy

Jeżeli chcemy zmodyfikować w pewien sposób wszystkie elementy naszej listy, to oczywiście możemy utworzyć funkcję, która w pętli będzie wykonywać, to co sobie wymyśliliśmy. Możemy jednak podejść do tego w inny sposób i zamiast pisania podobnego kodu w pętli dla różnych operacji użyć wskaźników na funkcje operujące na pojedynczym węźle i wywoływać tą funkcję w pętli.

Przykład

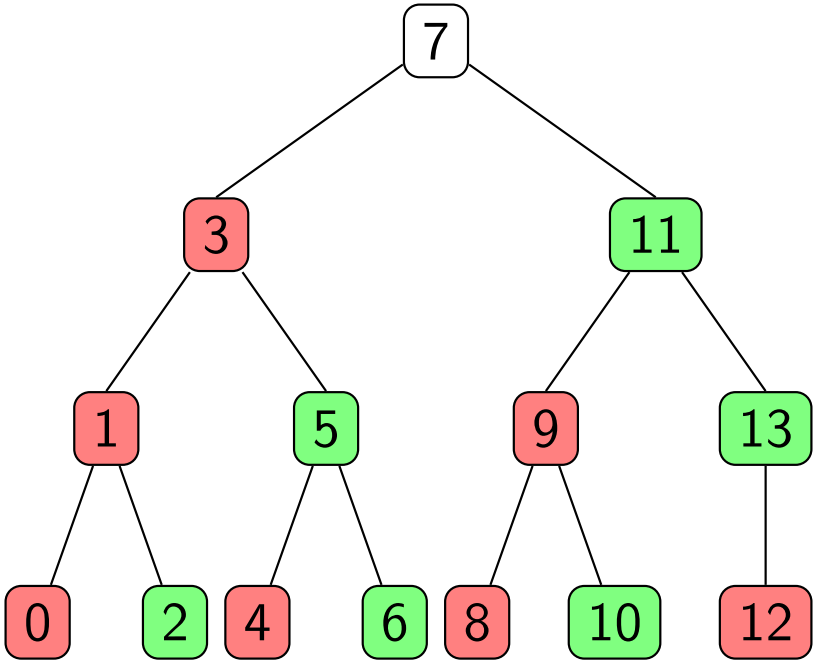
```
void calalista(student *pocz, void (*funkcja)(student *) ){
    for(;pocz!=0;pocz=pocz->next)
        funkcja(pocz);// nie może zmieniać wskaźników na następny
}

void zwiekszocene(student* aktualny){
    aktualny->ocena ++;
}
// gdzieś w kodzie
calalista(pierwszy,zwiekszocene);
```

Listy.c

Drzewa binarne

Drzewo binarne jest zbiorem elementów takim, że każdy element (węzeł) może być powiązany z maksymalnie dwoma innymi elementami (potomkowie lewy i prawy - wskaźniki). Jeżeli dany element nie zawiera potomków, to nazywamy go liściem. Element, od którego zaczyna się drzewo (czyli da się od niego dojść do każdego potomka) nazywamy korzeniem. Wysokością drzewa nazywamy maksymalną odległość od korzenia do liści.



Wśród drzew binarnych wyróżnia się drzewa zbalansowane - odległość każdego z liści jest co najwyżej o jeden mniejsza od wysokości drzewa. Wysokość drzewa zbalansowanego o n węzłach wynosi $[\log_2 n]$.
Na drzewach możemy wprowadzać pewne relacje porządku pomiędzy jego elementami zależne od danych zawartych w węzłach. Wyróżnia się tutaj:

- drzewa BST - mają tą własność, że lewy $<$ węzeł $<=$ prawy.
- stóg (kopiec) - każdy węzeł jest większy od swoich podwęzłów.

Wyróżniamy trzy sposoby rekurencyjnego przechodzenia całego drzewa:

- preorder - najpierw zajmujemy się wierzchołkiem, później przechodzimy do lewego poddrzewa, a na końcu do prawego;
- inorder - zaczynamy od przejścia do lewego poddrzewa, później zajmujemy się węzłem, a na końcu przechodzimy do prawego poddrzewa;
- postorder - zaczynamy od przejścia do lewego poddrzewa, później przechodzimy do prawego poddrzewa, a na końcu zajmujemy się węzłem;

Przykład

```
typedef struct drzewo{
    unsigned wartosc;
    struct drzewo *lewe;
    struct drzewo *prawe;
} drzewo;
drzewo *korzen;
```

Przykład - wysokość poddrzewa

```
#define max(x,y) x<y ? y: x;
unsigned wysokosc(drzewo *root){
    if(root==0) return 0;
    unsigned wyslewe=0,wysprawe=0;
    if(root->lewe) wyslewe=wysokosc(root->lewe)+1;
    if(root->prawe) wysprawe=wysokosc(root->prawe)+1;
    return max(wyslewe,wysprawe);
}// złożoność czasowa i pamięciowa O(n)
```

Przykład - zliczanie liści w poddrzewie

```
unsigned liscie(drzewo *root){
    if(root==0) return 0;
    if(root->lewe ==0 && root->prawe ==0)
        return 1;// gdy liść, zwracamy 1
    unsigned ilelisci=liscie(root->lewe);
    ilelisci+=liscie(root->prawe);
    return ilelisci;
}// złożoność czasowa i pamięciowa O(n)
```

Przykład - wykonywanie pewnej funkcji dla całego drzewa

```
void caledrzewo(drzewo *pocz, void (*funkcja)(drzewo *) ){
    if(pocz){
        caledrzewo(pocz->lewe,funkcja);
        funkcja(pocz);
        caledrzewo(pocz->prawe,funkcja);
    }
}

void zwiekszwartosc(drzewo* aktualny){
    aktualny->wartosc ++;
}
// gdzieś w kodzie
caledrzewo(korzen,zwiekszwartosc);
```

Przykład - wstawianie w drzewo BST

```
void wstaw(unsigned x,drzewo **root){
    drzewo *q = *root,*p =(drzewo*)malloc(sizeof(drzewo));
    if(p == 0) {
        printf("Brak pamięci!\n"); return;
    }
    *p = (drzewo) { x, 0, 0 };
    if(root==0){
        *root=p; return;
    }
    while(q){
        if( q->wartosc < x ) {// wstawiamy po lewej
            if( q->prawe == 0) {// doszliśmy do liścia to wstawiamy
                q->prawe = p; break;
            }
            q = q->prawe;// przechodzimy w prawo
        } else {
            if( q->lewe == 0){//doszliśmy do liścia to wstawiamy
                q->lewe = p; break;
            }
            q = q->lewe;// przechodzimy w lewo
        }
    }
}// takie wstawianie psuje drzewa zbalansowane i kopce
```

Przykład - usuwanie gałęzi i całego drzewa

```
void usunpoddrzewa(drzewo *root){
    if(root){
        if(root->lewe){
            usunpoddrzewa(root->lewe);// usunięcie lewej podgałęzi
            free(root->lewe);
            root->lewe=0;// wyzerowanie wskaźnika
        }
        if(root->prawe){
            usunpoddrzewa(root->prawe);// usunięcie prawej podgałęzi
            free(root->prawe);
            root->prawe=0;// wyzerowanie wskaźnika
        }
    }
}

void usundrzewo(){
    if(korzen){
        usunpoddrzewa(korzen);
        free(korzen);
        korzen=0;
    }
}
```

Przykład - wyszukiwanie nadwęzła w dowolnym drzewie

```
drzewo *WyszukajNadwezel(drzewo *aktualny,drzewo *root){
    if(root){
        drzewo *temp;
        if(root->lewe ==aktualny || root->prawe ==aktualny)
            return root;
        temp=Wyszukajnadwezel(aktualny,root->lewe);
        if(temp) return temp;
        temp=Wyszukajnadwezel(aktualny,root->prawe);
        if(temp) return temp;
    }
    return 0;
}
```

Drzewa.c