Zadanie 1 - Napisz program, który wczyta do pamięci całą zawartość ze standardowego wejścia, posortuje jej wiersze w pewien dziwaczny sposób, i wypisze efekt na standardowe wyjście.

Zakładamy, że wiersze kończą się pojedynczym znakiem `'\n'` (bądź `EOF`), a słowa składają się wyłącznie ze znaków "graficznych" (tj. drukowalnych, ale nie białych, użyj `isgraph` z `ctype.h`). Pozostałe znaki (tj. białe i kontrolne) rozdzielają słowa; w przetwarzaniu będziemy je ignorować, a na wyjściu zamiast ich bloków będziemy drukować pojedyncze spacje. Nie uwzględniamy pustych słów, za to uwzględniamy puste wiersze (kiedy pomiędzy dwoma znakami `'\n'` znajdują się tylko znaki białe i kontrolne, albo w ogóle żadne).

Kryterium porządkowania wierszy to zwykły porządek słownikowy, ale dla odwróconej kolejności słów w wierszu: wiersz `"aaa zzx"` jest późniejszy niż `"bbb yyz"`. Kolejność liter w słowach pozostaje niezmieniona, tj. składające się z jednego słowa wiersze `"abcd"` i `"dcba"` pozostają w "normalnym" porządku.

Żeby móc uporządkować wiersze według tego kryterium, a potem wydrukować je "normalnie", spamiętaj wejście w następującej strukturze:

- wiersz zareprezentuj jako tablicę napisów (słów); do Twojej decyzji należy, czy zapamiętasz też gdzieś jej długość, czy umieścisz w niej wartownika (np. `NULL`);
- całe wejście oczywiście zareprezentuj jako tablicę wierszy jw., będzie więc ona miała typ `char ***` (chyba że postanowisz wiersze poukrywać w jakichś `struct`ach).

Wczytywanie pojedynczego wiersza powinno odbywać się w osobnej funkcji; jej sygnatura jest do Twojej decyzji w zależności od potrzeb, ale jeśli będzie inna niż `char * readline()`, wyjaśnij znaczenie argumentów w komentarzu dokumentacyjnym. Może to być zresztą wskazane z praktycznych powodów – jeśli chcesz, by kolejne wywołania tej funkcji używały tego samego, już wcześniej zaalokowanego bufora itp., to należy go im jakoś przekazać. Nie rób żadnych założeń co do długości wczytywanych wierszy – jeśli brakuje Ci miejsca, zrealokuj bufor (np. prosząc o dwa razy więcej pamięci, niż było w nim dotychczas).

Do porównywania wierszy zgodnie z ww. porządkiem napisz komparator, który będzie zwracał –1, jeśli pierwszy argument jest wcześniejszy niż drugi, 1 jeśli jest późniejszy, i 0 jeśli są równe. Komparator może używać `strcmp`, ale lepiej wywołać ją wiele razy dla poszczególnych słów w porównywanych wierszach, zamiast sklejać te słowa w pojedynczy napis tylko po to, żeby wywołać `strcmp` raz.

Napisz plik nagłówkowy, w którym zadeklarujesz funkcję sortującą przyjmującą tablicę wierszy (czyli oczywiście wskaźnik na jej pierwszy element), ich liczbę, oraz wskaźnik na

funkcję porównującą. Plik z implementacją tej funkcji napisz (przy użyciu instrukcji warunkowych preprocesora) tak, by:

- przy kompilacji z flagą `-DUSE_QSORT` funkcja sortująca była tylko *wrapper*-em na `qsort` z `stdlib.h`, któremu przekaże swoje argumenty (oraz odpowiednią wartość rozmiaru komórki tablicy w trzeciej pozycji) – użyj do tego dyrektywy `#ifdef USE_QSORT`;
- w przeciwnym przypadku zaimplementuj algorytm prostego sortowania, który wykorzysta komparator przekazany jako argument; algorytm wybierz w zależności od reszty z dzielenia Twojego numeru indeksu przez 3 (0 – sortowanie bąbelkowe, 1 – sortowanie przez wstawianie, 2 – sortowanie przez wybór), umieść tę informację również w komentarzu.

Użycie `qsort` może wymusić zadeklarowanie komparatora z kwalifikatorem typu `const` dla argumentów, co w konsekwencji sprawi, że komparator nie będzie mógł modyfikować stanu swoich argumentów (ale i tak nie powinien, w końcu ma je tylko porównać, a nie zmieniać). Ten mechanizm będzie jeszcze omawiany na najbliższym wykładzie. M.in. dlatego warto najpierw napisać wersję z `qsort` (i dopasować wszystkie deklaracje), a dopiero potem zaimplementować własne sortowanie, żeby ono też było z nimi zgodne.

Proste sortowanie możesz spróbować zaimplementować tak, by jego sygnatura była równie "generyczna" jak dla `qsort` (wtedy warto to zrobić w kolejnej funkcji, a ww. funkcja będzie wyłącznie *wrapper*-em na jedno lub drugie sortowanie generyczne), ale gdyby były z tym podejściem jakieś problemy, możesz odłożyć to podejście na później. Zauważ, że jeśli nie zdążysz lub nie będziesz mieć ochoty zaimplementować prostego sortowania, to cała reszta zadania może być dalej implementowana i testowana dzięki kompilacji z `-DUSE_QSORT`.

Struktura logiczna całego programu powinna być taka:

- dopóki jest coś na wejściu:
    - wczytaj cały wiersz przy użyciu przeznaczonej do tego funkcji
    - podziel go na słowa i umieść w "głównej" tablicy, być może ją realokując, jeśli skończyło się w niej miejsce
- wywołaj funkcję sortującą, z "główną" tablicą i własnym komparatorem jako argumentami
- wypisz zawartość posortowanej "głównej" tablicy, wstawiając pojedyncze spacje pomiędzy słowa i pojedyncze znaki `'\n'` pomiędzy wiersze

Zarówno w funkcji wczytującej wiersze, jak i podziale wiersza na słowa może wystąpić niepowodzenie alokacji pamięci. W takiej sytuacji przerwij wczytywanie, a następnie posortuj i wypisz te wiersze, które udało się wczytać (natomiast pamięć zajęta przez ew.

ostatni, częściowo wczytany wiersz, powinna zostać od razu zwolniona). W trakcie wypisywania zwalniaj pamięć zajmowaną przez słowa i wiersze najwcześniej, jak się da (tj. od razu po ich wypisaniu).

```
2
3710728753390210279879799822083759024651013574025046376937677490009712648124896970078050417018260538
```

Wyjście

```
834842252
```

Przykład C

Wejście

```
5
3710728753390210279879799822083759024651013574025046376937677490009712648124896970078050417018260538
7432498619952474105947423330951305812372661730962991942213363574161572522430563301811072406154908250
230675882075393461711719803104210475137780632466763
```

Wyjście

```
272819012
```

Uwagi

Te liczby nie zmieszczą się w żadnym standardowym typie, zastosuj własny typ i zaimplementuj na nim operacje dodawania.

Zadanie 2 - Mamy otwarty i pełny klub oraz kolejkę przed klubem. Jeśli w klubie jest jeszcze miejsce, kolejne pełnoletnie osoby z kolejki są wpuszczane. Jeśli ktoś nie jest pełnoletni i byłaby jego kolej na wejście do klubu, opuszcza kolejkę. W czasie (na wejściu) następują kolejne zdarzenia:

- `+ <imię> <wiek>` - do kolejki przychodzi osoba o imieniu `<imię>` i wieku `<wiek>`,
- `- <imię>` - osoba o imieniu `<imię>` opuszcza kolejkę,
- `O X` - `X` osób opuszcza klub,
- `F <imię> <wiek> <imię_kolegi>` - do kolejki przychodzi osoba o imieniu `<imię>` i wieku `<wiek>`, która koleguje się z `<imię_kolegi>`, więc osoba o imieniu `<imię_kolegi>` wpuści ją przed siebie (załóż, że kolega zawsze jest w kolejce),
- `Z` - klub się zamyka.

Każda osoba ma unikatowe imię składające się z dużych i małych liter oraz cyfr. Długość imienia nie przekracza *1000* znaków. Wiek osób jest liczbą nie większą od *40*.

Możesz założyć, że na wejściu pojawi się nie więcej niż *10100* linii.

Wszystkie znaki na wejściu są z ASCII.

Zadanie: Wypisz imiona w kolejności w jakiej zostały wpuszczone do klubu

Wejście

Wejście składa się z linii postaci opisanej wyżej. Ostatnia linia wejścia zawiera pojedynczą literę `Z`.

Wyjście

Wyjście powinno składać się z tylu linii, ile osób wpuszczono do klubu przed zamknięciem. Każda linia wyjścia zawiera imię osoby.

Uwaga: **To zadanie ma limit pamięci 5 MB.**

Przykłady:

A

Wejście

```
+ Adam 20
+ Ola 23
+ Max 19
F Ala 24 Ola
+ Kamil 17
+ Iga 20
O 3
- Max
F Piotr 18 Kamil
O 2
+ Igor 30
Z
```

Wyjście:

```
Adam
Ala
Ola
Piotr
Iga
```

Wyjaśnienie:
Na początku w kolejce są Adam, Ola i Max. Potem za Adamem pojawia się Ala, a na końcu kolejki Kamil i Iga. Adam, Ala i Ola wchodzą do klubu, Max opuszcza kolejkę, a przed Kamilem pojawia się Piotr. Piotr i Iga wchodzą do klubu (Kamil ma urodziny jutro i stoi w kolejce bez szans na wejście. Gdy jest pierwszy, opuszcza kolejkę). Do kolejki dołącza Igor, ale klub się zamyka.

B

Wejście:

```
+ A 19
+ B 17
F c 20 B
- B
O 1
Z
```

Wyjście:

```
A
```

C

Wejście:

```
Z
```

Wyjście:

D

Wejście:

```
O 3
+ Adam 20
+ Bob 17
+ Cara 19
Z
```

Wyjście:

```
Adam
Cara
```